# Building Parallel Programs

SMPs, CLUSTERS, AND JAVA

ALAN KAMINSKY

# Mike Allgyer

## Spotlight on Careers in Computing

**Mike Allgyer has a Master's Degree in Computer Science from Rochester Institute of Technology, and is currently employed at Lockheed Martin as a software engineer.**

### Do you have any advice for students who are deciding between a career in industry or pursuing a more advanced degree?

If you can do it financially and you have the drive, go for another degree. Your career will always be there waiting. Many companies now offer tuition assistance as you pursue an advanced degree, so you need to decide if having your education paid for is worth working full-time and going to school at the same time. I went for a Master's right away and, while I accrued some debt, I'm glad I did.

### What's the most interesting project you've worked on in the past year?

My Master's project, a real-time ray tracer in CUDA, was interesting for many reasons. For one, it blended so many disciplines—3D graphics, physics, parallel computing, embedded systems—that I really got to stretch myself and tackle a significant project. Also, because CUDA was still relatively new at the time, it was exciting to be working on cutting-edge technology that not many people had much knowledge about.

### What drew you to your current field of specialization?

I always enjoyed computers and technology and doing weird mind puzzles as a kid, but for a long time I really didn't know what I wanted to do as far as a career. Then I took a few programming courses in high school, and as soon as I realized how fun it was to make a computer solve a problem for me, I was hooked. I also have a very visual and artistic side, so when I discovered 3D graphics, I had found my passion.

### Where do you see yourself in ten years?

I would love to own my own software company someday, but if that doesn't happen, I would just like to see myself being challenged and developing new technologies. If I'm helping create something that really blows people's hair back, regardless of whether it's for my own company or somebody else's, I think I'll be happy.

# Building Parallel Programs

## SMPs, Clusters, and Java

**Alan Kaminsky**

COURSE TECHNOLOGY
CENGAGE Learning™

Australia • Brazil • Japan • Korea • Mexico • Singapore • Spain • United Kingdom • United States

*This page intentionally left blank*

# Dedication

To my father, Edmund Kaminsky

COURSE TECHNOLOGY
CENGAGE Learning™

**Building Parallel Programs: SMPs, Clusters, and Java**
**Alan Kaminsky**

Managing Editor: Marie Lee

Acquisitions Editor: Amy Jollymore

Senior Product Manager: Alyssa Pratt

Editorial Assistant: Julia Leroux-Lindsey

Marketing Manager: Bryant Chrzan

Senior Content Project Manager: Jill Braiewa

Associate Content Project Manager: Lisa Weidenfeld

Art Director: Marissa Falco

Cover Designer: Riezebos Holzbaur Design Group

Cover Photo: © iStock Photography / Silense

Proofreader: Harry Johnson

Indexer: Liz Cunningham

Compositor: GEX Publishing Services

For product information and technology assistance, contact us at
**Cengage Learning Customer & Sales Support, 1-800-354-9706**

For permission to use material from this text or product, submit all requests online at **www.cengage.com/permissions**
Further permissions questions can be emailed to
**permissionrequest@cengage.com**

ISBN-13: 978-1-4239-0198-3

ISBN-10: 1-4239-0198-3

**Course Technology**
20 Channel Center Street
Boston, Massachusetts 02210
USA

The programs in this book are for instructional purposes only.

They have been tested with care, but are not guaranteed for any particular intent beyond educational purposes. The author and the publisher do not offer any warranties or representations, nor do they accept any liabilities with respect to the programs.

Cengage Learning is a leading provider of customized learning solutions with office locations around the globe, including Singapore, the United Kingdom, Australia, Mexico, Brazil, and Japan. Locate your local office at: **international.cengage.com/region**

Cengage Learning products are represented in Canada by Nelson Education, Ltd.

To learn more about Course Technology, visit **www.cengage.com/coursetechnology**

Purchase any of our products at your local college store or at our preferred online store **www.ichapters.com**

Printed in the United States of America
1 2 3 4 5 6 7 15 14 13 12 11 10 09

# Table of Contents

Chapter 11
## Parallel Image Generation

Chapter 12
## Load Balancing

Chapter 13
## Reduction

Chapter 14
## Parallel Random Number Generation

Chapter 15
## Reduction, Part 2

Chapter 16
## Sequential Dependencies

Chapter 17
## Barrier Actions

Chapter 18
## Overlapping

Part III
# Clusters

Chapter 19
## A First Cluster Parallel Program

Chapter 20

**Parallel Message Passing**

Chapter 21

**Massively Parallel Problems, Part 3**

Chapter 22

**Data Slicing**

Chapter 23

**Load Balancing, Part 2**

Chapter 24

**Measuring Communication Overhead**

Chapter 25

**Broadcast**

Chapter 26

**Reduction, Part 3**

Chapter 27

**All-Gather**

Chapter 28

**Scalability and Pipelining**

Chapter 29

**Overlapping, Part 2**

Chapter 30

**All-Reduce**

Chapter 38
**Phylogenetic Tree Construction**

**Appendices**

A
**OpenMP**

B
**Message Passing Interface (MPI)**

C
**Numerical Methods**

D
**Atomic Compare-and-Set**

# Preface

## 1 Scope

*Building Parallel Programs (BPP)* teaches the craft of designing and coding—building—parallel programs, namely programs that employ multiple processors running all at once to solve a computational problem in less time than on one processor (speedup), to solve a larger problem in the same time (sizeup), or both.

*BPP* covers techniques for parallel programming on both major categories of parallel computers, SMPs and clusters. A shared memory multiprocessor (SMP) parallel computer consists of several central processing units sharing a common main memory, all housed in a single box. The "dual-core" and "multicore" computers now available from most vendors are examples of SMP parallel computers. An SMP parallel computer is programmed using multiple threads running in one process and manipulating shared data in the main memory. A cluster parallel computer consists of several processors, each with its own (non-shared) memory, interconnected by a dedicated high speed network. A cluster parallel computer is programmed using multiple processes, one per processor; each process manipulates data in its own memory and exchanges messages with the other processes over the network. *BPP* also covers techniques for parallel programming on a hybrid SMP cluster parallel computer (a cluster each of whose members is an SMP computer), which requires both multithreading and message passing in the same program.

*BPP* relies heavily on studying actual, complete, working parallel programs to teach the craft of parallel programming. The programs in the book are written in Java and use a Java class library I developed, **Parallel Java** (http://www.cs.rit.edu/~ark/pj.shtml). The Parallel Java Library hides the low-level details of multithreading and message passing, allowing the programmer to write parallel programs using high-level constructs.

For each parallel program examined in *BPP*, the source code is included in the text, interspersed with explanatory narrative. Learning to program begins with studying programs; so resist the temptation to gloss over the source code, and give the same attention to the source code as to the rest of the text. The program source files are also included in the Parallel Java Library. By downloading the Library, the source files can be studied using a text editor or an integrated development environment.

A major emphasis in *BPP* is the use of performance metrics—running time, speedup, efficiency, sizeup—in the design of parallel programs. For each parallel program studied, the book reports the program's running time measurements on a real parallel computer with different numbers of processors, the book explains how to analyze the running time data to derive performance metrics, and the book explains how the metrics provide insights that lead to improving the program's design. Consequently, *BPP* covers such performance-related topics as cache interference, load balancing, and overlapping in addition to parallel program design and coding.

## 2 Rationale

Why write another book on parallel programming? Like many textbooks, *BPP* grew out of my dissatisfaction with the state of parallel computing in general and my dissatisfaction with existing parallel programming textbooks.

Some books emphasize the concepts of parallel computing or the theory of parallel algorithms, but say little about the practicalities of writing parallel programs. *BPP* is not a parallel algorithms text, and it is not primarily a parallel computing concepts text; rather, *BPP* is a parallel *programming* text. But to teach practical parallel programming requires using a specific programming language. I have chosen to use Java. Why write a book on parallel programming in Java?

Three trends are converging to move parallel computing out of its traditional niche of scientific computations programmed in Fortran or C. First, parallel computing is becoming of interest in other domains that need massive computational power, such as graphics, animation, data mining, and informatics; but applications in these domains tend to be written in newer languages like Java. Second, Java is becoming the principal programming language students learn, both in high school AP computer science curricula and in college curricula. Recognizing this trend, the ACM Java Task Force has recently released a collection of resources for teaching introductory programming in Java (http://jtf.acm.org). Third, even desktop and laptop personal computers are now using multicore CPU chips. In other words, today's desktop and laptop PCs are SMP parallel computers, and in the near future even "regular" applications like word processors and spreadsheets will need to use SMP parallel programming techniques to take full advantage of the PC's hardware. Thus, there is an increasing need for *all* computing students to learn *parallel* programming as well as regular programming; and because students are learning Java as their main language, there is an increasing need for students to learn parallel programming *in Java.* However, there are no textbooks specifically about parallel programming in Java. *BPP* aims to fill this gap.

In the scientific computing arena, parallel programs are generally written either for SMPs or for clusters. Reinforcing this dichotomy are separate standard libraries—OpenMP (http://openmp.org/wp/) for parallel programming on SMPs, MPI (http://www.mpi-forum.org/) for parallel programming on clusters. Because SMPs perform best on certain kinds of problems and clusters perform best on other kinds of problems, it is important to learn both SMP and cluster parallel programming techniques. Most books focus either on SMPs or on clusters; none cover both in depth. *BPP* covers both.

However, in my experience OpenMP and MPI are difficult to teach, both because they are large and intricate libraries and because they are designed for programming in non-object-oriented Fortran and C, not object-oriented Java with which most students are familiar. The Parallel Java Library provides the capabilities of OpenMP and MPI in a unified, object-oriented API. Using the Parallel Java Library, *BPP* teaches the same parallel programming concepts and patterns as OpenMP and MPI programs use, but in an object-oriented style using Java, which is easier for students to learn. OpenMP aficionados will recognize Parallel Java's thread teams, parallel regions, work-sharing parallel for loops, reduction variables, and other features. MPI devotees will recognize Parallel Java's communicators and its message passing operations—send, receive, broadcast, scatter, gather, reduce, and others. Having mastered the concepts and patterns, students can then more easily learn OpenMP itself or MPI itself should they ever need to write OpenMP or MPI programs. Appendix A and Appendix B provide brief introductions to OpenMP and MPI and pointers to further information about them.

As multicore machines become more common, hybrid parallel computers—clusters of multicore machines—will become more common as well. However, there are no standard parallel programming libraries that integrate multithreading and message passing capabilities together in a single library. Hybrid parallel programs that use both OpenMP and MPI are not guaranteed to work on every platform, because MPI implementations are not required to support multithreading. While hybrid parallel programs can be written using only the process-based message passing paradigm with MPI, sending messages between different processes' address spaces on the same SMP machine often yields poorer performance than simply sharing the same address space among several threads. A hybrid parallel program should use the shared memory paradigm for parallelism within each SMP machine and should use the message passing paradigm for parallelism between the cluster machines. I developed the Parallel Java Library especially for writing parallel programs of this kind.

Furthermore, not much has been published about hybrid parallel programming techniques, and to my knowledge there are no textbooks that cover these techniques in depth. *BPP* aims to fill this gap by including cutting-edge material on hybrid parallel programming. Even in "plain" clusters where each node has only one CPU, considerable synergy arises from combining message passing parallel programming with multithreaded parallel programming, and *BPP* covers these techniques as well.

Finally, many existing parallel programming textbooks give short shrift to real-world applications. Either the books only cover general parallel programming techniques and never mention specific real-world applications at all, or the books merely describe examples of real-world applications and never study the actual code for such applications. In contrast, *BPP* covers three real-world parallel program codes in depth, one computational medicine problem and two computational biology problems. These applications reinforce the general parallel programming techniques covered in the rest of the book and show how to deal with practical issues that arise when building real parallel programs.

## 3. Target Audience and Prerequisites

*BPP* is aimed at upper division undergraduate students and graduate students taking a first course in parallel computing. *BPP* is also suitable for professionals needing to learn parallel programming.

I assume you know how to write computer programs, in Java, as typically taught in the introductory computer science course sequence.

I assume you are familiar with computer organization concepts such as CPU, memory, and cache, as typically taught in a computer organization or computer architecture course. Chapter 2 reviews these concepts as applied to parallel computers.

I assume you are familiar with what a thread is and with the issues that arise in multithreaded programs, such as synchronization, atomic operations, and critical sections. This material is typically taught in an operating systems course. However, when writing Parallel Java programs, you never have to write an actual thread or use low-level constructs like semaphores. The Parallel Java Library does all that for you under the hood.

I assume you are familiar with computer networking and with the notion of sending data in messages between processors over a network, as typically taught in a data communications or computer networks course. I also assume you are familiar with object serialization in Java. Again, when writing Parallel Java programs, you never have to open a socket or send a datagram. The Parallel Java Library does it for you under the hood.

Finally, I assume you have a mathematical background at the level of first-year calculus. We will be doing some derivatives and logarithms as we model and analyze parallel program performance.

## 4. Organization

*BPP* is organized into five parts.

Part I covers preliminary material. Chapter 1 defines what parallel computing is and gives examples of problems that benefit from parallel computing. After a brief history of parallel computing, Chapter 2 covers the parallel computer hardware and software concepts needed to develop parallel programs effectively. Chapter 3 describes the general process of designing parallel programs, based on the design patterns of Carriero and Gelernter. Chapter 4 gives a gentle introduction to parallel programming with Parallel Java.

Part II is an in-depth study of parallel programming on SMP parallel computers. Chapters 5, 6, and 7 introduce massively parallel problems and the programming constructs used to solve them, notably parallel loops. Chapters 8, 9, and 10 focus on performance. Chapter 8 introduces parallel program performance metrics, specifically running time, speedup, efficiency, and experimentally determined sequential fraction. Anomalies in these metrics reveal the issue of cache interference, covered in Chapter 9 along with techniques for avoiding cache interference. Chapter 10 looks at two further performance metrics, sizeup and sizeup efficiency. Returning to parallel program design, Chapter 11 discusses the issues that arise when using a parallel program to generate images. Chapter 12 covers parallel problems that need load balancing to achieve good performance, along with parallel loop schedules for load balancing. Chapter 13 introduces the topic of parallel reduction using thread safe shared variables. After an interlude in Chapter 14 on how to generate random numbers in a parallel program, Chapter 15 continues the topic of parallel reduction using reduction operators and critical sections. Chapter 16 looks at problems with sequential dependencies and how to partition such problems among multiple threads. Chapter 17 covers the barrier action, a construct for interspersing sequential code within parallel code. Concluding the coverage of SMP parallel programming, Chapter 18 shows how to increase performance by overlapping computation with I/O.

Part III shifts the focus to parallel programming on cluster parallel computers. After a gentle introduction in Chapter 19 to cluster parallel programming with Parallel Java, Chapter 20 covers the fundamental concepts of message passing. Chapter 21 describes how to solve massively parallel problems that do not need to communicate any data. Turning to problems that require communicating large amounts of data, Chapter 22 shows how to slice data arrays and matrices into pieces and send the pieces in messages. Chapter 23 illustrates how to do load balancing in a cluster parallel program using the master-worker pattern, along with data slicing. Chapter 24 derives a mathematical model for a cluster parallel program's communication time. This is used in suceeding chapters to model the parallel program's running time, providing insight into the maximum performance the program can achieve. Chapter 24 also shows how to redesign a master-worker program to reduce the communication time. The next three chapters focus on three so-called "collective communication" message passing operations and examine programs that illustrate each operation: Chapter 25, broadcast; Chapter 26, reduce; Chapter 27, all-gather. Chapter 28 focuses on scalability, describing how to assess a parallel program's performance and memory requirements as it scales up to larger problem sizes. Chapter 28 also covers pipelining, a technique that allows problems with large memory requirements to scale up by adding processors to the cluster. Chapter 29 returns to the topic of overlapping, showing how to do overlapped computation and communication in a cluster parallel program. The next two chapters conclude Part III with three more collective communication operations: Chapter 30, all-reduce; Chapter 31, all-to-all and scan.

Part IV brings the shared memory and message passing paradigms together to write programs for hybrid parallel computers. Chapter 32 shows how to solve massively parallel problems with parallel programs containing both multiple processes and multiple threads per process. Chapter 33 covers load balancing on a hybrid parallel computer; the two degrees of freedom—load balancing among the processes, and load balancing among the threads within each process—provide considerable flexibility in the program design. Chapter 34 addresses the issues that arise when collective communication operations, specifically broadcast, must be done in a multithreaded message passing program. Chapter 35 covers parallel data set querying techniques, which are employed in many widely-used high performance scientific programs.

Parts II, III, and IV illustrate the general parallel programming techniques by studying the design, source code, and performance of several Parallel Java programs. These include programs to carry out a known plaintext attack on the Advanced Encryption Standard (AES) block cipher; compute an image of the Mandelbrot Set; estimate the value of π using a Monte Carlo technique; compute a histogram of the Mandelbrot Set image's pixels; find all shortest paths in a graph using Floyd's Algorithm; compute the evolution of a continuous cellular automaton; calculate the motion of antiprotons swirling around in a particle trap; calculate the temperature at each point on a metal plate by numerical solution of a partial differential equation; perform a Kolmogorov-Smirnov test on a random number generator; and compute the prime counting function using the Sieve of Eratosthenes. Many of the programs appear in several versions demonstrating different parallel programming techniques. Some of the programs appear in all three versions—SMP, cluster, and hybrid—to highlight the differences between the three paradigms. To emphasize the utility of parallel computing in domains other than the traditional ones of computational science and engineering, some of the problems solved with parallel programs in *BPP* are from "non-scientific" areas, such as cryptography and mathematics. However, the problems in Parts II, III, and IV were chosen solely for pedagogical reasons—to illustrate parallel programming techniques—and are not necessarily problems with any great significance in the real world.

Having covered parallel programming techniques in general, Part V goes on to solve three real-world problems requiring massive computation. Chapter 36 gives a cluster parallel program for magnetic resonance image (MRI) spin relaxometry, a computational medicine problem, based on curve fitting via nonlinear least squares. Chapter 37 gives two hybrid parallel programs for protein sequence querying, a computational biology problem, based on the Smith-Waterman local alignment algorithm. Chapter 38 gives an SMP parallel program for maximum parsimony phylogenetic tree construction, another computational biology problem, based on a parallel branch-and-bound algorithm.

Although many of the programs in *BPP* employ numerical algorithms, *BPP* is not a numerical methods textbook or a scientific computing textbook. The numerical methods are explained at a level sufficient to understand the parallel programs, but space does not permit detailed descriptions of the mathematics behind the numerical methods. To find out more about the numerical methods, see the references in the "For Further Information" section at the end of each chapter.

The book concludes with four appendices. Appendix A gives a brief introduction to OpenMP, describing OpenMP's features and comparing and contrasting OpenMP with Parallel Java. Appendix B does the same for MPI. Appendix A and Appendix B also compare the performance of OpenMP and MPI parallel programs written in C with the performance of the same parallel programs written in Java using the Parallel Java Library, demonstrating that Java programs can run as fast as or faster than C programs. Appendix C describes the numerical methods used throughout the book to analyze the parallel programs' performance. Appendix D covers the atomic compare-and-set operation, which can be used to achieve high-performance thread synchronization.

Exercises are included at the end of Parts I, II, III, and IV. Some are pencil-and-paper problems, some involve writing short parallel programs, some require investigating a program's behavior, some require a bit of research. I find that the best exercises integrate concepts from multiple chapters; thus, I have put the exercises at the end of each part of the book rather than at the end of each chapter.

## 5. Teaching

*BPP* contains more than enough material for a one-semester course on parallel computing. The book can also be used in a two-semester course sequence on parallel computing. The first semester would cover general SMP and cluster parallel programming techniques using Parts I–III. The second semester would cover hybrid parallel programming using Part IV and the real-world parallel programming applications in Part V, supplemented by the research literature and the instructor's own examples.

In the Computer Science Department at the Rochester Institute of Technology, I have been teaching the two-quarter parallel computing course sequence since 2005 using draft versions of the material in *BPP*. The "Parallel Computing I" course covers Parts I–III, the "Parallel Computing II" course covers Parts IV–V. I assign my students two term programming projects each quarter. In Parallel Computing I, the first project is to write an SMP parallel program to solve a stated problem; the second project is to write a cluster parallel program to solve the same problem. The students write a sequential and a parallel version of each program in Java using the Parallel Java Library, and they measure their programs' performance running on SMP and cluster parallel computers. In Parallel Computing II I do the same, except the second project is to write a hybrid parallel program.

Of course, to do parallel programming projects you will need a parallel computer. Nowadays it's easy to set up a parallel computer. Simply get a multicore server, and you have an SMP parallel computer. Or get several workstations and connect them with a dedicated high speed network like a 1-Gbps Ethernet, and you have a cluster parallel computer. Or do both. An interesting configuration is a hybrid parallel computer with four quad-core nodes. This gives you four-way parallelism for SMP parallel programs running with four threads on one node, sixteen-way parallelism for cluster parallel programs running with four processes on each of the nodes, and sixteen-way parallelism for hybrid parallel programs running with one process and four threads on each of the nodes. Larger multicore nodes and larger clusters will let you scale up your programs even further.

You will also need the Parallel Java Library. Parallel Java is free, GNU GPL licensed software and is available for download from my web site (http://www.cs.rit.edu/~ark/pj.shtml). The Library includes the Parallel Java middleware itself as well as all the parallel programs in this textbook, with source code, class files, and full Javadoc documentation. Parallel Java is written in 100% Java and requires only the Java 2 Standard Edition JDK 1.5 or higher to run on any machine; I have tested it on Linux and Solaris. Parallel Java is designed to be easy to install and configure; complete instructions are included in the Javadoc.

I am happy to answer general questions about the Parallel Java Library, receive bug reports, and entertain requests for additional features. Please contact me by email at ark@cs.rit.edu. I regret that I am unable to provide technical support, specific installation instructions for your system, or advice about configuring your parallel computer hardware.

## 6. Acknowledgments

I would like to thank my student Luke McOmber, who helped write the first version of the Parallel Java Library in 2005.

I would like to thank the students in my Parallel Computing I and Parallel Computing II classes, who have helped me debug and refine the Parallel Java Library and the material in *BPP*: Michael Adams, Justin Allgyer, Brian Alliet, Ravi Amin, Corey Andalora, Sheehan Anderson, Michael Attridge, Justin Bevan, Benjamin Bloom, Patrick Borden, John Brothers, Andrew Brown, Kai Burnett, Patrick Campagnano, Stephen Cavilia, Samuel Chase, Mahendra Chauhan, Pierre Coirier, Christopher Connett, Sean Connolly, Adam Cornwell, Eric Cranmer, Mason Cree, Daniel Desiderio, Derek Devine, Joseph Dowling, Guilhem Duché, Christopher Dunnigan, Timothy Ecklund, Andrew Elble, Peter Erickson, James Fifield, Jonathan Finamore, Gaurav Gargate, Dalia Ghoneim, Andres Gonzalez, Jason Greguske, Allison Griggs, Craig Hammell, Joshua Harlow, Charles Hart, Jacob Hays, Darren Headrick, Ryan Hsu, Leighton Ing, Nilesh Jain, Sean Jordan, Alban Jouvin, Glenn Katzen, Paresh Khatri, Dayne Kilheffer, John Klepack, Venkata Krishna Sistla, Michael Kron, Rachel Laster, Andrew LeBlanc, Joshua Lewis, Nicholas Lucaroni, Jon Ludwig, John MacDonald, Sushil Magdum, Jiong Mai, Daniel McCabe, Austin McChord, Sean McDermott, Michael McGovern, Eric Miedlar, David Mollitor, Kyle Morse, Adam Nabinger, Andrew Nachbar, Anurag Naidu, Paul Nicolucci, Brian Oliver, Shane Osaki, Derek Osswald, Robert Paradise, Kunal Pathak, Bryce Petrini, James Phipps, Michael Pratt, Paul Prystaloski, Michael Pulcini, Andrew Rader, Jessica Reuter, Aaron Robinson, Seth Rogers, Francis Rosa, Christopher Rouland, William Rummler, Omonbek Salaev, Mark Sanders, Josef Schelch, Bhavna Sharma, Michael Singleton, Amanda Sitterly, David Skidmore, Lukasz Skrzypek, Benjamin Solwitz, Brent Strong, Daniel Surdyk, Michael Szebenyi, Sakshar Thakkar, Jonathan Walsh, Borshin Wang, Kevin Watters, Jason Watt, Andrew Weller, Trevor West, Melissa Wilbert, Junxia Xu, and Andrew Yohn.

I would like to acknowledge Nan Schaller, who pioneered the RIT Computer Science Department's parallel computing courses. Nan also secured an NSF grant to acquire our first two SMP parallel computers. I would not have been able to develop the Parallel Java Library and the material in *BPP* if Nan had not paved the way.

I would like to thank the RIT Computer Science Department chairs, first Walter Wolf, then Paul Tymann, who procured capital funds for additional parallel computers to support teaching our parallel computing courses. As I write we have two 4-processor SMP parallel computers, two 8-processor SMP parallel computers, a 32-processor cluster parallel computer, and a 40-processor hybrid parallel computer with ten quad-core nodes. All of these use commodity off-the-shelf hardware. Our department's teaching facilities truly are "embarrassingly parallel."

I would like to thank Paul Austin and the Xerox Corporation. Paul was instrumental in obtaining a Xerox University Affairs grant for me which provided part of the funding for our hybrid parallel computer. Without this computer I would not have been able to develop and test the material on hybrid parallel programming that appears in Part IV.

I would like to thank our department's system administrator, James "Linus" Craig, and our hardware technician, Mark Stamer, for setting up our parallel computers, keeping them running, and for being generally supportive as I experimented with parallel computing middleware.

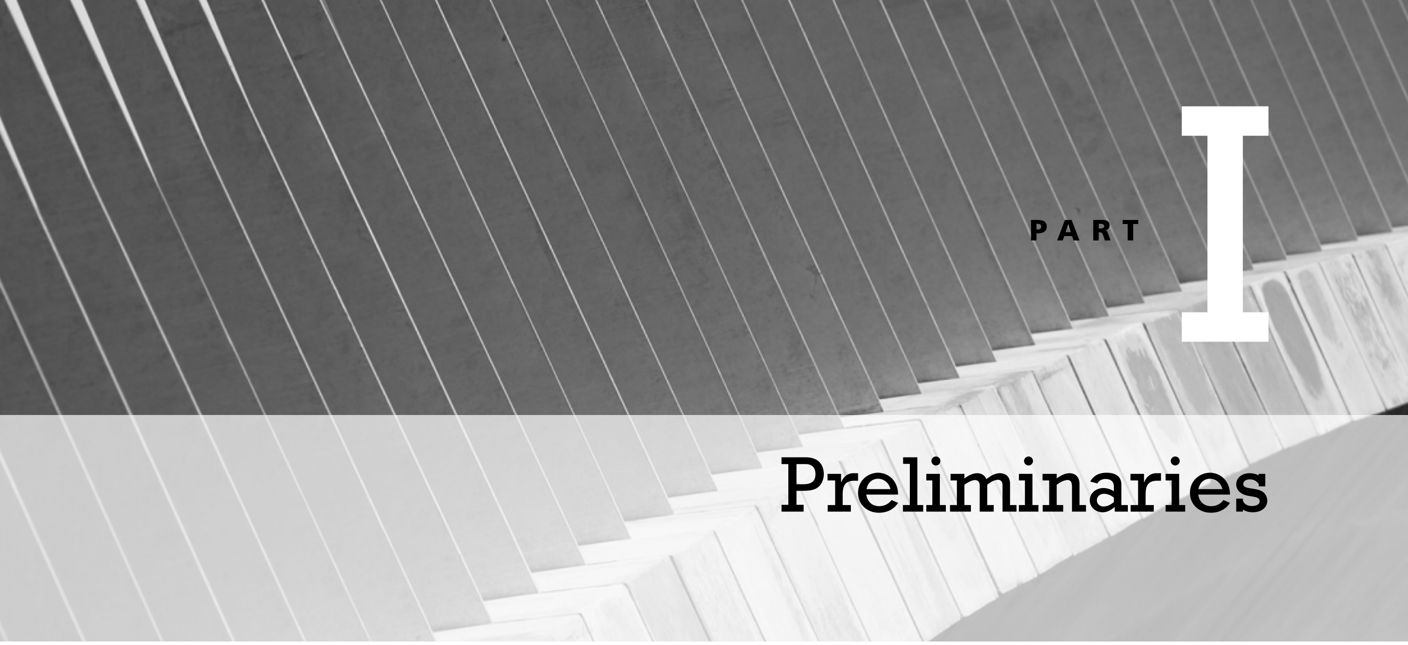
I would like to thank all the professionals at Course Technologies, especially Amy Jollymore, Alyssa Pratt, and Jill Braiewa, for their enthusiasm and effort in publishing the book. Without them you would not be holding *BPP* in your hands. I would also like to thank the reviewers who took the time to scrutinize the manuscript and offer many helpful suggestions: Paul Gray, University of Northern Iowa; David Hemmendinger, Union College; Lubomir Ivanov, Iona College; April Kontostathis, Ursinus College; Tom Murphy, Contra Costa College; George Rudolph, The Citadel; Jim Teresco, Mount Holyoke College; and George Thiruvathukal, Loyola University Chicago. *BPP* is a better book because of these folks' efforts.

Lastly, I would like to thank my wife Margaret and my daughters Karen and Laura. Many were the days and evenings I would disappear into my upstairs sanctum to write. Thank you for your love and support during this long project.

I dedicate this book to my father, Edmund Kaminsky. Dad, you've waited a long time for this. Thank you for all your encouragement.

Alan Kaminsky
December 2008

*This page intentionally left blank*



# PART I

# Preliminaries

*This page intentionally left blank*

# CHAPTER 1

# Parallel Computing

in which we discover what parallel computing is; we learn how it can help us solve computational problems; and we survey several problems that benefit from parallel computing

# CHAPTER 1 Parallel Computing

## 1.1 Bigger Problems, Faster Answers

Many of today's computer applications require massive amounts of computation. Here's an example of a computational medicine problem involving **magnetic resonance imaging (MRI)**. MRI is a technique for making images of the inside of an organism, such as a living person's brain, without cutting the patient open. The MRI scanner sends a brief, high-intensity radio frequency pulse through the patient. The pulse reverses the orientation of the spins of the atoms in the patient. The MRI scanner then measures the atomic spins in a two-dimensional plane, or slice, through the subject as the atoms "relax," or return to their normal spin orientations. Each measurement takes the form of an *N*×*N*-pixel image. The MRI scanner takes a sequence of these image snapshots at closely spaced time intervals (Figure 1.1).

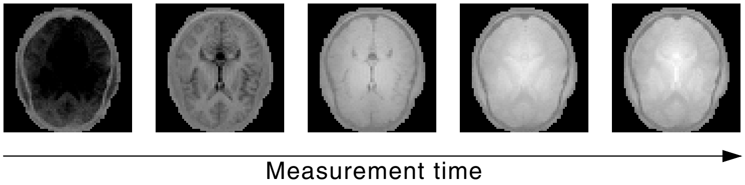


**Figure 1.1** Sequence of magnetic resonance images for one slice of a brain

The rates at which the atoms' spins relax helps a doctor diagnose disease. In healthy tissue, the spins relax at certain rates. In diseased tissue, if abnormal chemicals are present, the spins relax at different rates. The sequence of measured spin directions and intensities for a given pixel can be analyzed to determine the spin relaxation rates in the tissue sample corresponding to that pixel. Such a **spin relaxometry analysis** requires sophisticated and time-consuming calculations on each pixel's data sequence to recover the underlying spin relaxation rates from the typically imperfect and noisy images. (In Chapter 36, we will examine MRI spin relaxometry analysis in more detail.)

One computer program that did the spin relaxometry analysis took about 76 seconds to do the calculations for a single pixel. To analyze all the pixels in, say, a 64×64-pixel image, a total of 4,096 pixels, would take about 311,000 seconds—over 3.5 days—of computer time. And this is for just one slice through the subject. A complete MRI scan involves *many* slices to generate a three-dimensional picture of the subject's interior. Clearly, the calculations need to be completed in a drastically shorter time for spin relaxometry analysis to be a useful diagnostic technique.

One alternative for reducing the calculation time is to get a faster computer. The earlier-noted timing measurement was made on a computer that was several years old. Running the same program on an up-to-date computer that is 5 times faster than the original computer would take about 62,000 seconds to analyze a 64×64-pixel image, or about 17 hours instead of 3.5 days.

It used to be that computer clock speeds doubled roughly every two years. However, that trend finally may be ending. By 2004, CPU chips had achieved clock speeds in the 3 GHz range. If the trend had continued, clock speeds should have reached 12 GHz by 2008—but they did not. Instead, in late 2004, chip makers started moving away from the strategy of boosting chip performance by increasing the raw clock speed, opting instead to introduce architectural features such as "hyperthreaded" and "multicore" chips (we will say more about these later). Although clock speeds do continue to increase, in the future there is little hope left for dramatically reduced calculation times from faster chips.

Another alternative for reducing the calculation time is to switch to a faster algorithm. Suppose we could devise a different program that was 100 times faster than the original; then running on the faster computer, it would take about 620 seconds—about 10 minutes—to analyze a 64×64-pixel image. Putting it another way, the calculation time per pixel would be 0.15 seconds instead of 76 seconds. However, algorithmic improvements can take us only so far. There comes a point where the fastest-known algorithm on the fastest available computer is still simply not fast enough.

A third alternative is *to have several computers working on the problem simultaneously*. Say we have $K$ computers instead of just one. We divide the problem up into $K$ pieces and assign one piece to each computer. For the MRI spin relaxometry analysis problem, each computer analyzes $(N \times N)/K$ pixels. Then all the computers go to work on their respective pieces at the same time. We say that the computers are executing **in parallel**, and we refer to the whole conglomeration as a **parallel computer**. Henceforth we will refer to the individual units within the parallel computer as **processors** to distinguish them from the parallel computer as a whole.

We could apply a parallel computer to the MRI spin relaxometry analysis problem in either of two ways. Suppose we have a 16-processor parallel computer. We divide the 4,096 pixels among the 16 processors. Because each processor has to do the calculations for only 256 pixels, and because all the processors are running at once, the computation takes only 256 × 0.15 seconds, or about 39 seconds, instead of 10 minutes. The parallel computer let us *reduce the running time* by a factor of 16 while holding the problem size constant. Used in this way, a $K$-processor parallel computer ideally gives a **speedup** of $K$ (Figure 1.2).

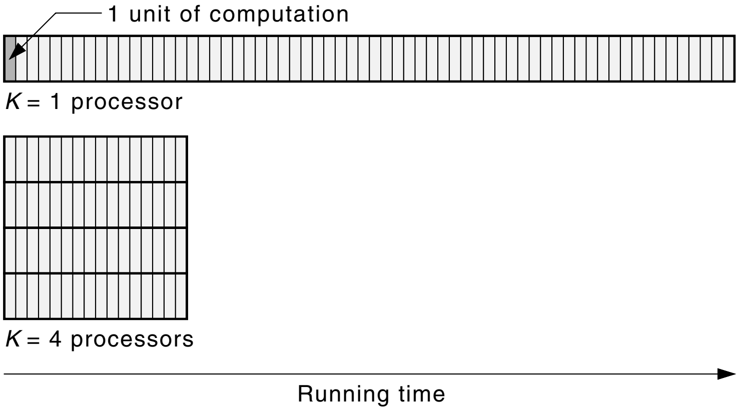


**Figure 1.2** Speedup with a parallel computer—same problem size, $(1/K)$ × running time

On the other hand, suppose we use our 16-processor parallel computer to analyze a magnetic resonance image with 16 times as many pixels—a 256×256-pixel image, which is actually the typical size of a magnetic resonance image used for a medical diagnosis. Either the image encompasses a larger area, or the image covers the same area at a finer resolution. Then the computation still takes the same 10 minutes, but we have analyzed a larger image. The parallel computer has let us *increase the problem size* by a factor of 16 while holding the running time constant. Used in this way, a *K*-processor parallel computer ideally gives a size increase, or **sizeup**, of *K* (Figure 1.3).

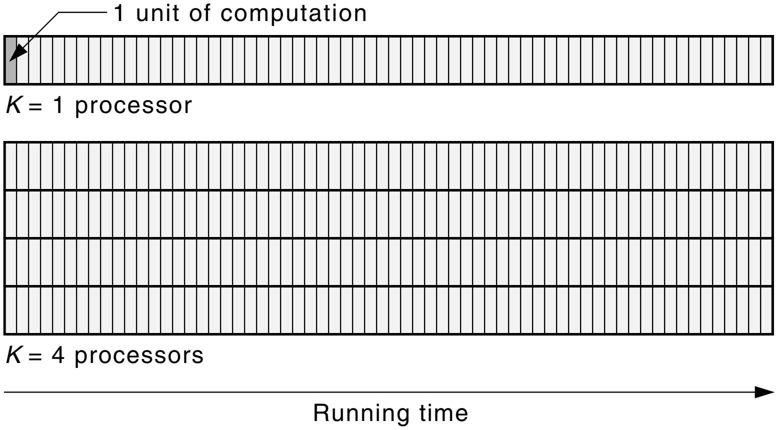


**Figure 1.3** Sizeup with a parallel computer—*K* × problem size, same running time

Of course, we can employ both strategies. A 64-processor parallel computer, for example, would let us analyze a 256×256-pixel image in one-fourth the time it takes a single computer to analyze a 64×64-pixel image. The more processors we add, the bigger the images we can analyze, and the faster we can get the answers.

To sum up, **parallel computing** is the discipline of employing multiple processors running all at once to solve the same problem in less time (speedup), to solve a larger problem in the same time (sizeup), or both. Another term often used is **high-performance computing (HPC)**, which emphasizes the improved performance parallel computing provides in solving larger problems or solving problems faster.

## 1.2 Applications for Parallel Computing

In 2004, the U.S. Office of Science and Technology Policy released a report titled “Federal Plan for High-End Computing.” This report lists four broad application areas—climate and weather, nanoscale science and technology, life sciences, and aerospace vehicle design—with problems requiring massive amounts of computation, that can and do benefit from parallel computing. Problems in other areas, such as astrophysics, mathematics, games, and animation, are also attacked using parallel computing. Here are a few examples of such problems:

**Weather forecasting**. In August 2005, Hurricane Katrina devastated the U.S. Gulf Coast, flooding the city of New Orleans, killing more than 1,800 people, and causing $100 billion in damage. Computer models of the atmosphere, such as the Weather Research and Forecasting (WRF) Model, can predict a storm's track (the path it will follow, where it will hit land) and intensity (wind speed). The WRF program takes a portion of the earth's atmosphere—a given region of the earth's surface, up to a given height above the surface—and divides this three-dimensional region into many small 3-D cells. The WRF program then uses physical models to calculate atmospheric conditions in each cell, as influenced by the neighboring cells, and advances the forecast in a series of many small time steps. The WRF program uses parallel computing to handle the calculations for large numbers of cells and time steps.

Accurate hurricane track and intensity forecasts can help officials decide how to prepare for a storm and where to evacuate if necessary. However, current hurricane forecasting programs are not all that accurate. With current models, track forecast errors can be as large as 70 kilometers (km), and wind speed forecast errors can be as large as 37 kilometers per hour (kph). A wind-speed shift of 37 kph can change a mere tropical storm to a Category 3 major hurricane. A track error of 70 km could cause officials to evacuate Boca Raton, Florida when they should have evacuated Miami.

To get more accurate predictions, the cells and the time steps in the model must be made smaller; this means that the model must include *more* cells and *more* time steps to cover the same geographic region and the same time span. For example, if the cell's dimensions are decreased by a factor of 2, the number of cells must increase by a factor of 8 to cover the same 3-D region. If the time step size is also decreased by a factor of 2, the number of time steps must increase by a factor of 2 to cover the same time span. Thus, the total amount of computation goes up by a factor of 16. This in turn means that even more powerful parallel computers and parallel programs will be needed to calculate the models.

**Climate modeling**. On what date will the rainy season begin in Brazil this year, so farmers will know when to plant their crops? Why is rainfall decreasing in the Indian subcontinent—could it be caused by pollution from burning wood for fuel and cooking? What effect will increased levels of atmospheric carbon dioxide have on the world's climate—none at all, or drastic warming that will melt the polar ice caps and inundate coastal cities? Computer-based climate models can answer these questions (and fan the controversies). The Community Climate System Model (CCSM), for example, models the atmosphere, ocean, sea ice, and land surface using a three-dimensional grid of cells like the WRF model. The CCSM program runs on a parallel computer to simulate the earth's climate over the entire globe for time spans of thousands of years. Because there is no end to the number of climatological features and the level of detail that can be included in climate simulation programs, such programs will continue to need the power of parallel computers well into the future.

**Protein sequence matching**. Imagine you are a biochemist. You have isolated a new protein from the creature you are studying, but you have no idea what the protein does—could it be an anti-cancer agent? Or is it just a digestive enzyme?

One way to get a clue to the protein's function is to match your protein against other proteins; if your protein closely matches proteins of known function, chances are your protein does something similar to the matching proteins. Chemically, a protein is a group of amino acids linked together into a long string. Twenty different amino acids are found in proteins, so a protein can be represented as a string of letters from a 20-character alphabet; this string is called the protein's "sequence." Protein sequence databases collect information about proteins, including their sequences and functions. The Swiss-Prot database, for example, contains well over 385,000 protein sequences ranging in length from 2 to 35,000 amino acids, with a median length of around 300 amino acids. You can determine your new protein's sequence and match it against the protein sequences in the database. Doing so is more complicated than looking up a credit card number in a financial database, however. Rather than finding a single, exact match, you are looking for multiple, inexact but close matches.

The Basic Local Alignment Search Tool (BLAST) program is at present the premier tool for solving the preceding protein sequence matching problem. The BLAST program combines a "local alignment" algorithm, which matches a piece of one protein sequence against a piece of another protein sequence, with a search of all protein sequences in the database. Because local alignment is a computationally intensive algorithm and because the databases are large, parallel versions of BLAST are used to speed up the searches. In Chapter 37, we will design a parallel program for protein sequence database searching.

**Quantum computer simulation**. A quantum computer exploits quantum mechanical effects to perform large amounts of computation with small amounts of hardware. A quantum computer register with $n$ qubits (quantum bits) can hold $2^n$ different states at the same time via "quantum superposition" of the individual qubits' states. Performing one operation on the quantum register updates all $2^n$ states simultaneously, making some hitherto intractable algorithms practical. For example, in 1994, Peter Shor of AT&T Bell Laboratories published a quantum algorithm that can factor large composite numbers efficiently. If we could do that, we could break the RSA public key cryptosystem, which is the basis for secure electronic commerce on the Internet. The potential for solving problems with polynomial time algorithms on a quantum computer—that would otherwise require exponential time algorithms on a classical computer—has sparked interest in quantum algorithms.

Although small, specialized quantum computers have been built, it will be quite some time before useful general-purpose quantum computers become available. Nonetheless, researchers are forging ahead with quantum algorithm development. Lacking actual quantum computers to test their algorithms, researchers turn to quantum computer simulators running on classical computers. The simulators must do massive amounts of computation to simulate the quantum computer's exponentially large number of states, making quantum computer simulation an attractive area for (classical) parallel computing. Several parallel simulator programs for quantum computers have been published.

**Star cluster simulation**. Astrophysicists are interested in the evolution of star clusters and even entire galaxies. How does the shape of the star cluster or galaxy change over time as the stars move under the influence of their mutual gravitational attraction? Many galaxies, including our own Milky Way, are believed to have a supermassive black hole (SMBH) at the center. How does the SMBH move as the comparatively much-lighter stars orbit the galactic center? What happens when two galaxies collide? While it's unlikely for individual stars in the galaxies to collide with each other, the galaxies as a whole might merge, or they might pass through each other but with altered shapes, or certain stars might be ejected to voyage alone through intergalactic space.

There are theories that purport to predict what will happen in these scenarios. But because of the long time scales involved, millions or billions of years, there has been no way to test these theories by observing actual star clusters or galaxies. So astrophysicists have turned to observing the evolution of star clusters or galaxies *simulated in the computer*. In recent years, "computational astrophysics" has revolutionized the field and revealed a host of new phenomena for theorists to puzzle over.

The most general and powerful methods for simulating stellar dynamics, the so-called "direct $N$-body methods," require enormous amounts of computation. The simulation proceeds as a series of time steps. At each time step, the gravitational force on each star from all the other stars is calculated, each star's position and velocity are advanced as determined by the force, and the cycle repeats. A system of $N$ stars requires $O(N^2)$ calculations to determine the forces. To simulate, say, one million stars requires

$10^{12}$ force calculations—*on each and every time step*; and one simulation may run for thousands or millions of time steps. In Chapter 27, we will examine an *N*-body problem in more detail.

To run their simulations, computational astrophysicists turn to special purpose hardware. One example is the GRAPE-6 processor, developed by Junichiro Makino and his colleagues at the University of Tokyo. (GRAPE stands for GRAvity piPE.) The GRAPE-6 processor is a parallel processor that does only gravitational force calculations, but does them much, much faster than even the speediest general-purpose computer. Multiple GRAPE-6 processors are then combined with multiple general-purpose host processors to form a massively parallel gravitational supercomputer. Examples of such supercomputers include the GRAPE-6 system at the University of Tokyo and the "gravitySimulator" system built by David Merritt and his colleagues at the Rochester Institute of Technology.

**Mersenne primes**. Mersenne numbers—named after French philosopher Marin Mersenne (1588–1648), who wrote about them—are numbers of the form $2^n–1$. If a Mersenne number is also a prime number, it is called a Mersenne prime. The first few Mersenne primes are $2^2–1$, $2^3–1$, $2^5–1$, $2^7–1$, and $2^{13}–1$. (Most Mersenne numbers are not prime.) The largest known Mersenne prime is $2^{43,112,609}–1$, a whopper of a number with nearly 13 million decimal digits, discovered in August 2008 by the Great Internet Mersenne Prime Search (GIMPS) project.

Starting in 1996, the GIMPS project has been testing Mersenne numbers to find ever-larger Mersenne primes. The GIMPS project uses a "virtual parallel computer" to test candidate Mersenne numbers for primality in parallel. The GIMPS parallel computer consists of PCs and workstations contributed by volunteers around the globe. Each volunteer downloads and installs the GIMPS client program on his or her computer. The client runs as a lowest-priority process, and thus uses CPU cycles only when the computer is otherwise idle. The client contacts the GIMPS server over the Internet, obtains a candidate Mersenne number to work on, subjects the candidate to an array of tests to determine whether the candidate is prime, and reports the result back to the server. Because the candidate Mersenne numbers are so large, the primality tests can take days to weeks of computation on a typical PC. Since commencing operation, the GIMPS project has found 12 previously unknown Mersenne primes with exponents ranging from 1,398,269 to the aforementioned 43,112,609.

While Mersenne primes have little usefulness beyond pure mathematics, there is a practical incentive for continuing the search. The Electronic Frontier Foundation (EFF) has announced a $100,000 prize for the first discovery of a ten-million-digit prime number—a prize now claimed by the GIMPS project. The EFF has also announced further prizes of $150,000 for the first 100-million-digit prime number and $250,000 for the first one-billion-digit prime number. While any prime number (not necessarily a Mersenne prime) qualifies for the prizes, the GIMPS project has perhaps the best chance at reaching these goals as well. According to their Web site, with these prizes, "EFF hopes to spur the technology of cooperative networking and encourage Internet users worldwide to join together in solving scientific problems involving massive computation."

**Search for extraterrestrial intelligence (SETI)**. Since 1997, researchers at the University of California, Berkeley have been using the radio telescope at Arecibo, Puerto Rico to search for signs of extraterrestrial intelligence. As the telescope scans the sky, the researchers record radio signals centered around a frequency of 1.42 GHz. (Because hydrogen atoms throughout the universe emit energy at this frequency, a fact of which any advanced civilization ought to be aware, SETI researchers feel that extraterrestrials who want to announce their presence would broadcast signals near this frequency.) These recorded signals are then analyzed to determine if they contain any "narrowband" signals—signals confined to a small frequency range. Whereas most of the energy in the signal is “broadband” background noise spread out over a wide frequency range, a narrowband signal—like an AM radio, FM radio, television, or satellite signal—is more likely to have been generated by an intelligent being. Any detected narrowband signals are subjected to further scrutiny to eliminate signals of terrestrial origin. Signals that cannot be identified as terrestrial might just be extraterrestrial.

Because of the enormous amounts of radio signal data collected and the extensive computations needed to analyze the data to detect narrowband signals, the Berkeley researchers realized they needed a massively parallel computer. Rather than buy their own parallel supercomputer, they used the same approach as the GIMPS project and created the SETI@home project in 1999. Volunteers install the SETI@home client program on their computers. Running as a screen saver or as a low-priority background process, each client program contacts the SETI@home server over the Internet, downloads a “workunit” of radio signal data, analyzes the workunit, and sends the results back to the server. The SETI@home virtual supercomputer has analyzed nearly 300 million workunits so far, each workunit occupying 350 kilobytes of data and requiring 10 to 12 hours of computation on a typical PC, and has detected over 1.1 billion narrowband signals.

The SETI@home approach to parallel computing was so successful that the Berkeley researchers developed the Berkeley Open Infrastructure for Network Computing (BOINC), a general framework for Internet-based client-server parallel computation. Volunteers can donate compute cycles to any project that uses the BOINC infrastructure. Some 50 projects now use BOINC; they range from SETI@home itself to protein structure prediction to climate modeling.

Has SETI@home found any signs of extraterrestrial intelligence? The researchers are still working their way through the 1.1 billion narrowband signals that have been detected. So far, none can be conclusively stated as being of extraterrestrial origin.

**Chess**. On May 11, 1997, Garry Kasparov, chess grandmaster and then world chess champion, sat down opposite IBM chess computer Deep Blue for a six-game exhibition match. Kasparov and Deep Blue had met 15 months earlier, on February 10, 1996, and at that time Kasparov prevailed with three wins, two draws, and one loss. After extensive upgrades, Deep Blue was ready for a rematch. This time, the results were two wins for Deep Blue, one win for Kasparov, and three draws, for an overall score of Deep Blue 3.5, Kasparov 2.5. It was the first time a computer had won a match against a reigning human world chess champion. After his defeat, Kasparov demanded a rematch, but IBM refused and retired the machine. Deep Blue’s two equipment racks are now gathering dust at the National Museum of American History and the Computer History Museum.

Deep Blue was a massively parallel, special-purpose supercomputer, consisting of 30 IBM RS/6000 SP server nodes augmented with 480 specialized VLSI chess chips. It chose chess moves by brute force, analyzing plays at a rate of 200 million chess positions per second. It also had an extensive database of opening positions and endgames.

While Deep Blue is gone, computer chess research continues. The focus has shifted away from specialized parallel hardware to software programs running on commodity parallel machines. In November 2006, the Deep Fritz chess program, a parallel program running on a PC with two Intel dual-core CPU chips, beat then world chess champion Vladimir Kramnik with two wins and four draws. Many people now believe that in the realm of chess, human dominance over computers is at its end. Of course, it’s not really man versus machine, it’s human chess players against human computer builders and programmers.

**Animated films**. In 1937, Walt Disney made motion picture history with the animated feature film *Snow White and the Seven Dwarfs*. While Disney had been making animated short films since the 1920s, *Snow White* was the world's first *feature-length* animated film. In those days each frame of the film was laboriously drawn and colored by hand on celluloid. All that would change in 1995, when Pixar Animation Studios and Walt Disney Pictures released *Toy Story*, the world's first feature-length *computer*-animated film. Six years later, in 2001, DreamWorks Animation SKG debuted the computer-animated film *Shrek*, the first animated film to win an Academy Award. Since then, scarcely a year has gone by without several new feature-length computer-animated film releases.

During the early stages of production on a computer-animated film, the artists and designers work mostly with individual high-end graphics workstations. But when the time comes to "render" each frame of the final film, adding realistic surface textures, skin tones, hair, fur, lighting, and so on, the computation shifts to the "render farm"—a parallel computer with typically several thousand nodes, each node a multicore server. For *Toy Story*, the render farm had to compute a 77-minute film, with 24 frames per second and 1,536×922 pixels per frame—more than 157 billion pixels altogether. Despite the render farm's enormous computational power, it still takes hours or even days to render a single frame. As movie audiences come to expect ever-more-realistic computer-animated films, the render farms will continue to require increasingly larger parallel computers running increasingly sophisticated parallel rendering programs.

We've only scratched the surface, but perhaps you've gotten a sense of the broad range of problems that are being solved using parallel computing. The largest and most challenging of today's computer applications rely on parallel computing. It's an exciting area in which to work!

## 1.3 For Further Information

On the end of the trend towards ever-increasing CPU clock speeds:

- Herb Sutter. A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30(3):16–22, March 2005.
- Craig Szydlowski. Multithreaded technology and multicore processors. *Dr. Dobb's Journal*, 30(5):58–60, May 2005.

On applications for high performance computing:

- U.S. Office of Science and Technology Policy. Federal plan for high-end computing. May 10, 2004.
  http://www.nitrd.gov/pubs/2004_hecrtf/20040702_hecrtf.pdf

On hurricane forecasting and the Weather Research and Forecast Model:

- Thomas Hayden. Super storms: No end in sight. *National Geographic*, 210(2):66–77, August 2006.

- J. Michalakes, J. Dudhia, D. Gill, T. Henderson, J. Klemp, W. Skamarock, and W. Wang. The Weather Research and Forecast Model: software architecture and performance. In *Proceedings of the 11th ECMWF Workshop on the Use of High Performance Computing In Meteorology*, 2004. http://www.wrf-model.org/wrfadmin/docs/ecmwf_2004.pdf
- The Weather Research and Forecasting Model. http://www.wrf-model.org/index.php

On using parallel computing for weather modeling. “The Weather Man,” a fascinating science fiction story written in 1962 when computers were still young, computer networks not yet invented, and parallel computers barely beginning, nonetheless managed to give a remarkably prescient depiction of parallel computing on what are now known as networked workstation clusters:

- Theodore L. Thomas. “The Weather Man.” *Analog*, June 1962.

“The Weather Man” is reprinted in a couple more recent science fiction collections:

- Isaac Asimov and Martin H. Greenberg, editors. *Isaac Asimov Presents the Great SF Stories #24 (1962)*. DAW Books, 1992.
- David G. Hartwell and Kathryn Cramer, editors. *The Ascent of Wonder: The Evolution of Hard SF*. Tor Books, 1994.

On the Community Climate System Model:

- Special issue on the Community Climate System Model. *Journal of Climate*, 19(11), June 1, 2006.
- W. Collins, C. Bitz, M. Blackmon, G. Bonan, C. Bretherton, J. Carton, P. Chang, S. Doney, J. Hack, T. Henderson, J. Kiehl, W. Large, D. McKenna, B. Santer, and R. Smith. The Community Climate System Model Version 3 (CCSM3). *Journal of Climate*, 19(11):2122–2143, June 1, 2006.
- Community Climate System Model. http://www.ccsm.ucar.edu/

On protein sequence matching:

- The Universal Protein Resource (UniProt), including the Swiss-Prot protein sequence database. http://www.uniprot.org/
- National Center for Biotechnology Information. http://www.ncbi.nlm.nih.gov/
- S. Altschul, W. Gish, W. Miller, E. Myers, and D. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215(3):403–410, October 5, 1990.

On quantum computers, factoring algorithms, and simulators:

- D. DiVincenzo. Quantum computation. *Science*, 270(5234):255–261, October 13, 1995.
- C. Williams and S. Clearwater. *Ultimate Zero and One: Computing at the Quantum Frontier*. Copernicus, 2000.
- S. Loepp and W. Wootters. Protecting Information: *From Classical Error Correction to Quantum Cryptography*. Cambridge University Press, 2006.
- P. Shor. Algorithms for quantum computation: discrete logarithms and factoring. In *Proceedings of the 35th Annual Symposium on the Foundations of Computer Science*, 1994, pages 124–134.
- K. Obenland and A. Despain. A parallel quantum computer simulator. arXiv preprint arXiv:quant-ph/9804039v1, April 1998. http://arxiv.org/abs/quant-ph/9804039v1
- J. Niwa, K. Matsumoto, and H. Imai. General-purpose parallel simulator for quantum computing. arXiv preprint arXiv:quant-ph/0201042, January 2002. http://arxiv.org/abs/quant-ph/0201042v1
- K. De Raedt, K. Michielsen, H. De Raedt, B. Trieu, G. Arnold, M. Richter, T. Lippert, H. Watanabe, and N. Ito. Massive parallel quantum computer simulator. *Computer Physics Communications*, 176(2):121–136, January 15, 2007.

On gravitational supercomputers at the University of Tokyo and the Rochester Institute of Technology:

- J. Makino, T. Fukushige, M. Koga, and K. Namura. GRAPE-6: massively-parallel special-purpose computer for astrophysical particle simulations. *Publications of the Astronomical Society of Japan*, 55(6):1163–1187, December 2003.
- S. Harfst, A. Gualandris, D. Merritt, R. Spurzem, S. Portegies Zwart, and P. Berczik. Performance analysis of direct *N*-body algorithms on special-purpose supercomputers. *New Astronomy*, 12(5):357–377, July 2007.

On GIMPS and the EFF prime number prizes:

- The Great Internet Mersenne Prime Search. http://www.mersenne.org/
- Electronic Frontier Foundation Cooperative Computing Awards. http://www.eff.org/awards/coop.php

On SETI@home:

- E. Korpela, D. Werthimer, D. Anderson, J. Cobb, and M. Lebofsky. SETI@home—massively distributed computing for SETI. *Computing in Science and Engineering*, 3(1):78–83, January 2001.
- D. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer. SETI@home: an experiment in public-resource computing. *Communications of the ACM*, 45(11):56–61, November 2002.
- SETI@home. http://setiathome.berkeley.edu/

On BOINC:

- D. Anderson. BOINC: a system for public-resource computing and storage. In *Proceedings of the Fifth IEEE/ACM International Workshop on Grid Computing*, 2004, pages 4–10.
- Berkeley Open Infrastructure for Network Computing. http://boinc.berkeley.edu/

On Deep Blue:

- F. Hsu, M. Campbell, and J. Hoane. Deep Blue system overview. In *Proceedings of the Ninth International Conference on Supercomputing (ICS'95)*, 1995, pages 240–244.
- F. Hsu. *Behind Deep Blue: Building the Computer that Defeated the World Chess Champion*. Princeton University Press, 2002.
- F. Hsu. Chess hardware in Deep Blue. *Computing in Science and Engineering*, 8(1):50–60, January 2006.

On rendering *Toy Story*:

- T. Porter and G. Susman. Creating lifelike characters in Pixar movies. *Communications of the ACM*, 43(1):25–29, January 2000.

On Pixar's and DreamWorks' render farms:

- M. Hurwitz. Incredible animation: Pixar's new technologies. November 2004. http://www.uemedia.net/CPC/vfxpro/article_10806.shtml
- S. Twombly. DreamWorks animation artists go over the top with HP technology. May 2006. http://www.hp.com/hpinfo/newsroom/feature_stories/2006/06animation.html

# CHAPTER 2

# Parallel Computers

in which we briefly recount the history of parallel computers; we examine the characteristics of modern parallel computer hardware that influence parallel program design; and we introduce modern standard software libraries for parallel programming

# CHAPTER 2 Parallel Computers

## 2.1 A Brief History of Parallel Computers

To understand how modern parallel computers are built and programmed, we must take a quick look at the history of parallel computers.

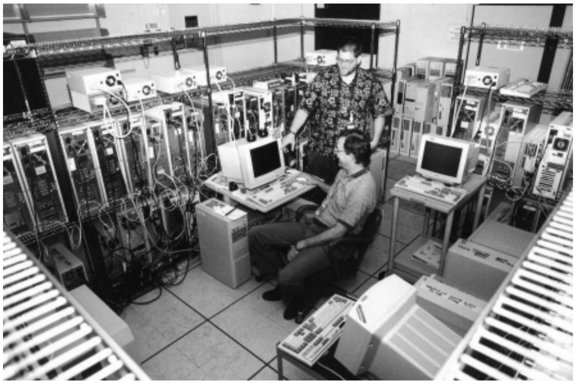
Up until the mid-1990s, there were no widely adopted standards in either parallel computer hardware or software. Many vendors sold parallel computers, but each vendor had its own proprietary designs for CPUs and CPU interconnection networks. Along with its proprietary hardware, each vendor provided its own proprietary languages and tools for writing parallel programs on its hardware—sometimes a variant of a scientific programming language such as Fortran, sometimes another language.

Because the hardware and software were mostly vendor-specific, parallel programs of that era tended not to be portable. You settled on a hardware vendor, then you used the vendor's supported parallel programming language to write your parallel programs. If you wanted to change vendors, you were faced with the dismaying prospect of rewriting and re-debugging all your programs using the new vendor's programming language.

As the twentieth century drew toward its close, four events took place that signaled the beginning of a paradigm shift for parallel computing. First, in the late 1970s, Robert Metcalfe and David Boggs invented the Ethernet local area network at the Xerox Palo Alto Research Center; in 1980, the 10-Mbps Ethernet standard was published by Digital Equipment Corporation, Intel, and Xerox; in 1983, a variation was standardized as IEEE Std 802.3. Second, in 1981, the Internet Protocol (IP) and the Transmission Control Protocol (TCP), developed by Vinton Cerf and Robert Kahn, were published as Internet standards—"Request For Comments" (RFC) 791 and RFC 793. Third, also in 1981, IBM started selling the IBM PC, whose open architecture became the de facto standard for personal computers. By 1983, for the first time in the history of computing, there was an open standard computer hardware platform (the PC), an open standard local area network (Ethernet) for interconnecting computers, and an open standard network protocol stack (TCP/IP) for exchanging data between computers. Fourth, in 1991, Linus Torvalds released the first version of the Linux operating system, a free, Unix-like operating system for the PC, that included an implementation of TCP/IP.

In 1995, Thomas Sterling, Donald Becker, and John Dorband at the NASA Goddard Space Flight Center, Daniel Savarese at the University of Maryland, and Udaya Ranawake and Charles Packer at the Hughes STX Corporation published a paper titled "Beowulf: a parallel workstation for scientific computation." In this paper, they described what is now called a cluster parallel computer, built from off-the-shelf PC boards, interconnected by an off-the-shelf Ethernet, using the Internet standard protocols for communication, and running the Linux operating system on each PC. The Beowulf cluster's performance was equal to that of existing proprietary parallel computers costing millions of dollars, but because it was built from off-the-shelf components, the Beowulf cluster cost only a fraction as much.

A year later, William Hargrove and Forrest Hoffman at Oak Ridge National Laboratory proved that a parallel computer could be built for zero dollars. Lacking a budget to buy a new parallel computer, they instead took obsolete PCs that were destined for the landfill, loaded them with the free Linux operating system, and hooked them up into a Beowulf cluster. Dubbing their creation the "Stone SouperComputer" (Figure 2.1), after the fable of the soldier who cooked up "stone soup" from a small stone along with a bit of this and a bit of that contributed by the villagers in the story, eventually Hargrove and Hoffman had a cluster with 191 nodes.



http://www.extremelinux.info/stonesoup/photos/1999-05/image01.jpg

**Figure 2.1** The Stone Souper Computer

Once the PC, Ethernet, and TCP/IP became standardized, prices were driven down by mass production of microprocessor, memory, and Ethernet chips and cutthroat competition in the PC market. Today, computers are commodities you can buy in a store just like you can buy toasters and televisions—a state of affairs undreamed of when proprietary designs held sway. To get a parallel computer with a given amount of computing power, it usually costs much less to buy a multicore PC server, or to buy a bunch of PCs and a 1-Gbps Ethernet switch, from the corner computer store than to buy a proprietary computer; and because Linux is free, it doesn't cost anything to equip these PCs with an operating system. While proprietary parallel computer companies are still in business, their computers primarily occupy a niche at the ultra-high-performance end of the spectrum. The majority of parallel computing nowadays takes place on commodity hardware.

Since 1993, the TOP500 List (at http://www.top500.org) has been tracking the 500 fastest supercomputers in the world, as measured by a standard benchmark (the LINPACK linear algebra benchmark). In the June 1993 TOP500 List, only 104 of the top 500 supercomputers (21%) used commodity CPU chips—Intel i860s and Sun SuperSPARCs. The rest of the top 500 (79%) used proprietary CPUs. In contrast, the June 2008 TOP500 list shows 498 of the top 500 supercomputers using commodity CPU chips from (in alphabetical order) AMD, IBM, and Intel. By the way, in June 1993, 97 of the top 500 supercomputers were still single-CPU systems. In June 2008, *all* the top 500 supercomputers were parallel computers of one kind or another, with anywhere from 80 to over 200,000 processors.

In the transition to commodity hardware, parallel programming also shifted away from using proprietary programming languages to using standard, non-parallel programming languages, chiefly Fortran, C, and C++, coupled with standard parallel programming libraries. The Parallel Virtual Machine (PVM) library for programming cluster computers was first released in 1989. The Message Passing Interface (MPI) standard, also for programming cluster computers, was first released in 1994. The OpenMP standard for multithreaded parallel programming was first released in 1997. Like Linux, free versions of PVM, MPI, and OpenMP are widely available. Because these are hardware-independent standards, parallel programs written on one machine can be easily ported to another machine. The majority of parallel programming nowadays is done using a standard language and library.

Because parallel computing with commodity hardware and software is now firmly established, this book will focus on building parallel programs for commodity parallel computers. To build good parallel programs requires understanding the characteristics of parallel computer hardware and software that influence parallel program design. Next, we'll look at these characteristics and at the prevalent parallel computer architectures.

## 2.2 CPU Hardware

To help achieve the goal of ever-increasing computer performance (and sales), **central processing units (CPUs)** have become bewilderingly complex. A modern CPU may employ architectural features such as these:

- **Pipelined architecture**. While one instruction is being fetched from memory, another already-fetched instruction is being decoded, and several more already-decoded instructions are in various stages of execution. With multiple instructions in process at the same time in different stages of the pipeline, the CPU's effective computation speed increases.
- **Superscalar architecture**. The CPU has several functional units, each capable of executing a different class of instructions—an integer addition unit, an integer multiplication unit, a floating-point unit, and so on. If several instructions use different functional units and do not have other dependencies among each other, the CPU can execute all the instructions at once, increasing the CPU's effective speed.
- **Instruction reordering**. To utilize the CPU's pipeline and functional units to the fullest, the CPU may issue instructions in a different order from the order they are stored in memory, provided the results turn out the same.

High-level programming languages hide most of this architectural complexity from the programmer. The hardware itself, perhaps aided by the high-level language compiler or (in the case of Java) the virtual machine, takes care of utilizing the CPU's architectural features. A Java or C program, for example, does not have to be rewritten if it is going to be run on a superscalar CPU rather than a CPU with just one functional unit. However, there are two CPU architectural features that *do* make a difference in the way high-level language programs are written: cache memories, and symmetric multiprocessors.

**Cache memories**. As CPU speeds outpaced memory speeds, computer architects added a **cache memory** to avoid making a fast CPU wait for a slow main memory (Figure 2.2).

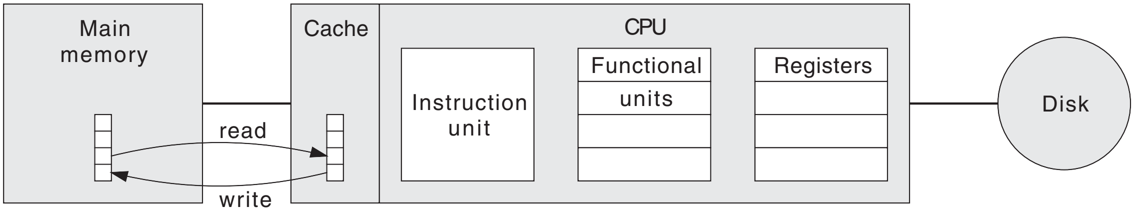


**Figure 2.2** Computer with cache memory

The main memory is large, typically gigabytes or hundreds of megabytes, but slow. The cache memory is much smaller than main memory, typically a few megabytes, but is also much faster than main memory. That is, it takes much less time for the CPU to read or write a word in cache memory than in main memory.

With the cache in place, when the CPU reads a word at a certain address, the CPU first checks whether the desired word has been loaded into the cache. If it has not—a **cache miss**—the CPU loads the entire **cache line** containing the desired word from main memory into the cache; then the CPU reads the desired word from the cache. The cache line size depends on the processor; 64 to 128 bytes is typical. Thereafter, when the CPU reads the same word, or reads another word located in the same cache line, there is a **cache hit**; the CPU reads the word directly from the cache and does not need to load it from main memory. Thus, if the **cache hit ratio**—the fraction of all memory accesses that are cache hits—is large, the CPU will read data words at nearly the speed of the fast cache memory and will seldom have to wait for the slow main memory.

As the CPU continues to load cache lines from main memory into the cache, eventually the cache becomes full. At the next cache miss, the CPU must replace one of the previously loaded cache lines with the new cache line. The cache's **replacement policy** dictates which cache line to replace. Various replacement policies are possible, such as replacing a *randomly chosen* cache line, or replacing the cache line that has not been accessed for the longest amount of time (the *least recently used* cache line).

When the CPU writes a word at a certain address, the CPU must load the relevant cache line from main memory if there is a cache miss, then the CPU can replace the contents of the desired word in the cache with the new value. Subsequent reads of that word will retrieve the new value from the cache. However, after a write, the contents of the word in the cache do not match the contents of the word in main memory; the cache line is said to be **dirty**. The cache's **write policy** dictates what to do about a dirty cache line. A *write-through cache* copies the dirty cache line to main memory immediately. A *write-back cache* copies the dirty cache line to main memory only when the cache line is being replaced.

The cache has a profound effect on program performance. A program's **working set** consists of all the memory locations the program is currently accessing, both locations that contain instructions and locations that contain data. If the program's working set fits entirely within the cache, the CPU will be able to execute the program as fast as it possibly can. This is often possible when the bulk of the program's time is spent in a tight loop operating on a data structure smaller than the cache. To the extent that the program's working set does not fit in the cache, the program's performance will be reduced. In this case, the name of the game is to design the program to minimize the number of inevitable cache misses.

Some CPUs are so fast that even the cache is a performance bottleneck. Such CPUs use a **multilevel cache** (Figure 2.3).

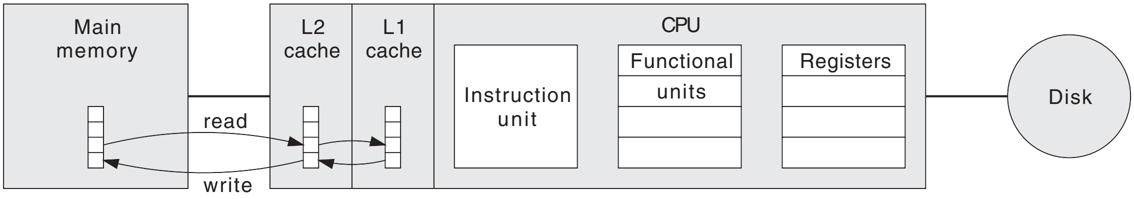


**Figure 2.3** Computer with multilevel cache

A **level-1 (L1) cache** sits between the CPU and the **level-2 (L2) cache** (formerly the only cache). The L1 cache is even faster than, and smaller than, the L2 cache. The L1 cache bears the same relationship to the L2 cache as the L2 cache bears to main memory. From the programmer's point of view, whether the cache has multiple levels is less important than the cache's existence. The name of the game is still to design the program to minimize the number of cache misses.

**Symmetric multiprocessors**. To achieve performance gains beyond what is possible on a single CPU, computer architects replicated the CPU, resulting in a **symmetric multiprocessor** (Figure 2.4). It is called "symmetric" because all the processors are identical. Each processor is a complete CPU with its own instruction unit, functional units, registers, and cache. All the processors share the same main memory and peripherals. The computer achieves increased performance by running multiple threads of execution simultaneously, one on each processor.

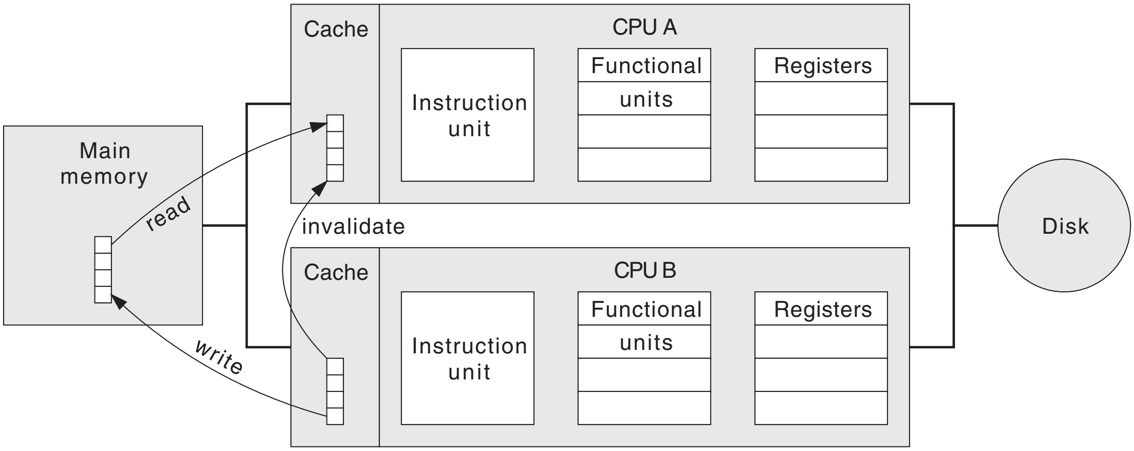


**Figure 2.4** Symmetric multiprocessor

Going to multiple processors, however, affects the caches' operation. Suppose CPU A reads the word at a certain address *x,* so that a copy of the relevant cache line is loaded from main memory into CPU A's cache. Suppose CPU B now writes a new value into the word at address *x*. CPU B's cache line is written back to main memory. However, CPU A's cache line no longer has the correct contents. The CPUs use a **cache coherence protocol** to bring the caches back to a consistent state. One popular cache coherence protocol uses **invalidation**. When CPU B writes its value into address *x*, CPU B sends an "invalidate" signal to tell CPU A that the contents of address *x* changed. In response, CPU A changes its cache state to say that the cache line containing address *x* does not reside in the cache. This is called "invalidating" the cache line. The next time CPU A reads address *x*, CPU A will see that the cache line containing address *x* is not loaded, CPU A will reload the cache line from main memory, and CPU A will retrieve the value written by CPU B.

Note that writing a word in one CPU can slow down another CPU, because the other CPU has to reread the word's cache line from slow main memory. This **cache interference** effect can reduce a parallel program's performance, and we will return to the topic of cache interference in Chapter 9.

Symmetric multiprocessor computers originally used a separate CPU chip for each processor (thread). However, as the number of transistors that could fit on a chip continued to increase, computer architects started to contemplate running more than one thread on a single chip. For example, instead of simultaneously issuing multiple instructions from a single instruction stream—a single thread—to multiple functional units, why not issue multiple instructions from *multiple* threads simultaneously to the functional units? Doing this for, say, two threads requires two instruction units, one to keep track of each thread's instruction stream. The result is called a **hyperthreaded CPU** (Figure 2.5).

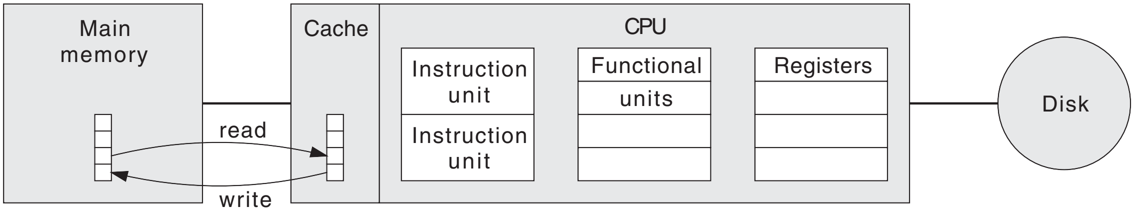


**Figure 2.5** Computer with hyperthreaded CPU

Two instruction units are by no means the limit. Some of today's fastest supercomputers use **massively multithreaded processors (MMPs)** that can handle hundreds of simultaneous threads. As soon as one thread stalls, perhaps because it has to wait for data to be loaded from main memory into the cache, the CPU can instantly switch to another thread and keep computing.

As transistor densities continued to increase, it became possible to replicate the whole processor, not just the instruction unit, on a single chip. In other words, it became possible to put a symmetric multiprocessor on a chip. Such a chip is called a **multicore CPU**. Alternatively, the name may refer to the number of processors on the chip: a **dual-core CPU**; a **quad-core CPU**; and so on. Multicore chips can themselves be aggregated into symmetric multiprocessor systems, such as a four-processor computer comprising two dual-core CPU chips.

It used to be that you could increase an application program's performance simply by trading in your old PC for a new one with a faster CPU chip. That won't necessarily work any longer. Now that chips have become hyperthreaded or multicore, your new PC may very well have a multicore CPU with the same clock speed as, or even a slower clock speed than, your old PC. If your application is single-threaded, as many are, it can run only on one CPU of the multicore chip and thus may run *slower* on your new PC! Until applications are redesigned as multithreaded programs—*parallel* programs—users will not see much of a performance improvement when running applications on the latest multicore PCs.

Having examined the salient features of individual CPUs, we next look at different ways to combine multiple CPUs to make a complete parallel computer.

## 2.3 SMP Parallel Computers

There are three principal parallel computer architectures using commodity hardware: SMPs, clusters, and hybrids. There is also a variant called a computing grid. Parallel coprocessors using commodity graphics chips have also been introduced.

A **shared memory multiprocessor (SMP)** parallel computer is nothing more than a symmetric multiprocessor system (Figure 2.6). Each processor has its own CPU and cache. All the processors share the same main memory and peripherals.

A parallel program running on an SMP parallel computer (Figure 2.7) consists of one process with multiple threads, one thread executing on each processor. The process's program and data reside in the shared main memory. Because all threads are in the same process, all threads are part of the same **address space**, so all threads access the same program and data. Each thread performs its portion of the computation and stores its results in the shared data structures. If the threads need to communicate or coordinate with each other, they do so by reading and writing values in the shared data structures.

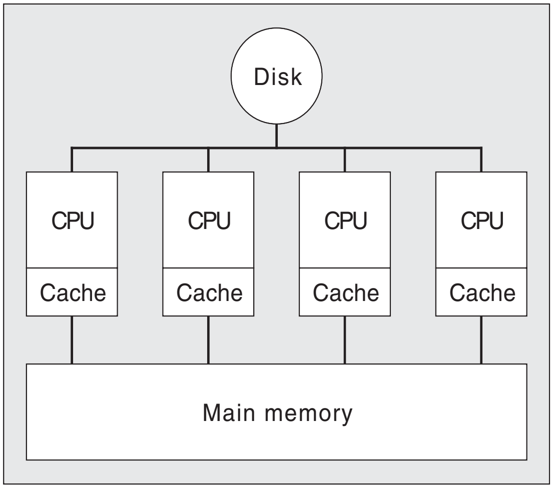


**Figure 2.6** SMP parallel computer

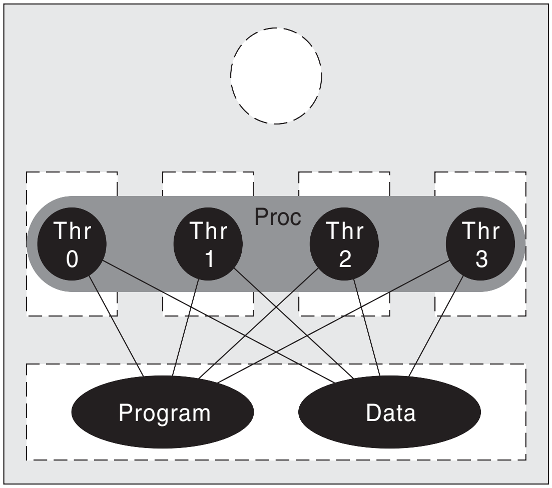


**Figure 2.7** SMP parallel program

## 2.4 Cluster Parallel Computers

A **cluster** parallel computer consists of multiple interconnected processor nodes (Figure 2.8). There are several **backend processors** that carry out parallel computations. There is also typically a separate **frontend processor**; users log into the frontend to compile and run their programs. There may be a shared file server. Each backend has its own CPU, cache, main memory, and peripherals, such as a local disk drive. Each processor is also connected to the others through a dedicated high-speed **backend network**. The backend network is used only for traffic between the nodes of the cluster; other network traffic, such as remote logins over the Internet, goes to the frontend. Unlike an SMP parallel computer, there is no global shared memory; each backend can access only its local memory. The cluster computer is said to have a **distributed memory**.

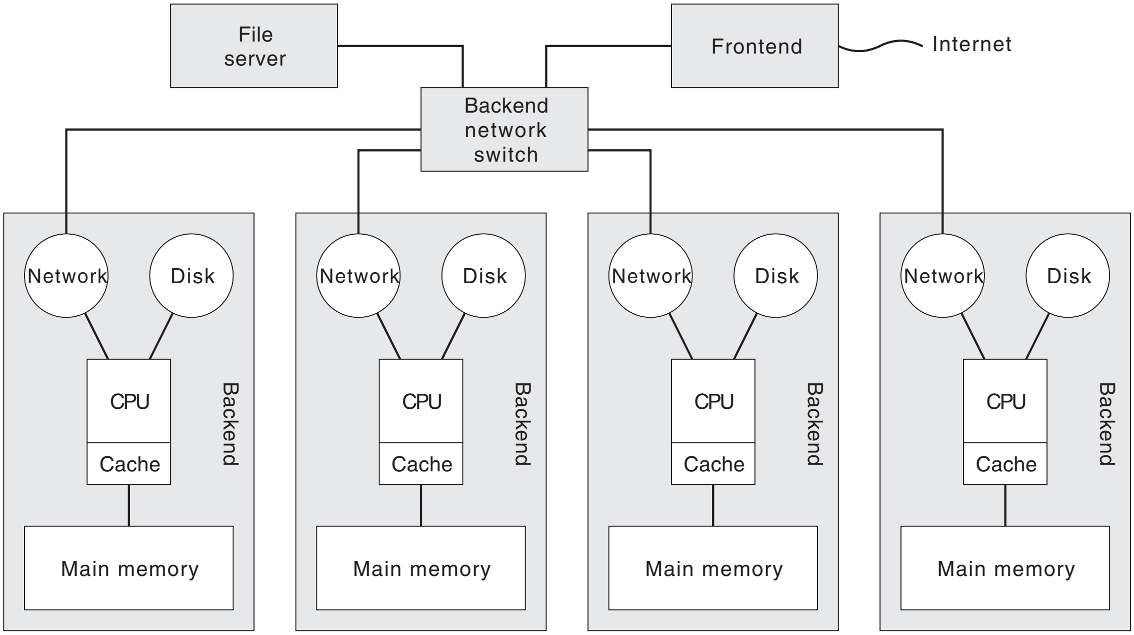


**Figure 2.8** Cluster parallel computer

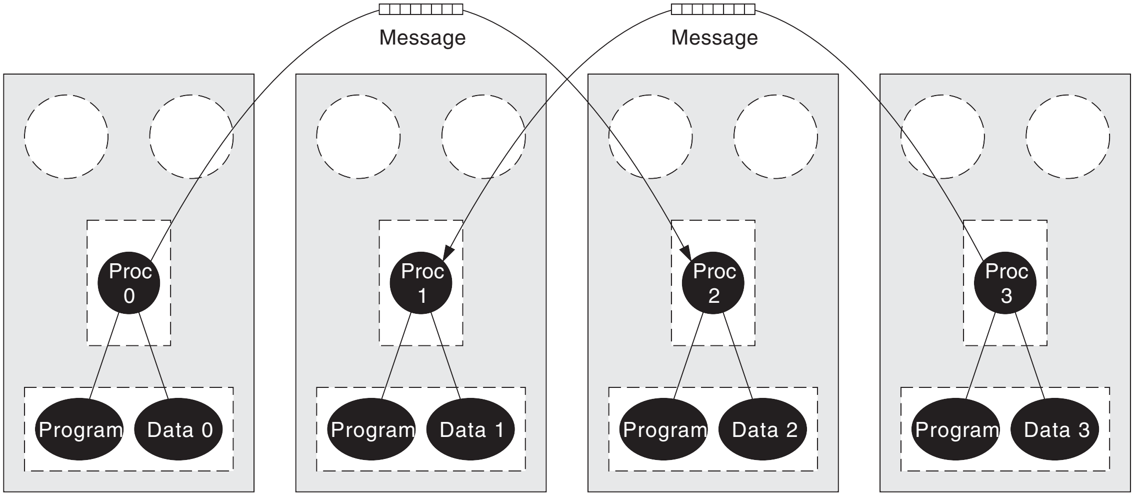


**Figure 2.9** Cluster parallel program

A parallel program running on a cluster parallel computer (Figure 2.9) consists of multiple processes, one process executing on each backend processor. Each process has its own, separate address space. All processes execute the same program, a copy of which resides in each process's address space in each backend's main memory. The program's data is divided into pieces; a different piece resides in each process's address space in each backend's main memory. Each process performs its portion of the computation and stores its results in the data structures in its own local memory. If one process needs a piece of data that resides in another process's address space, the process that owns the data sends a **message** containing the data through the backend network to the process that needs the data. The processes may also exchange messages to coordinate with one another, without transferring data. Unlike an SMP parallel program where the threads can simply access shared data in the one process's address space, in a cluster parallel program, the processes must be explicitly coded to send and receive messages.

To run a parallel program on a cluster, you typically log into the cluster's frontend processor and launch the program in a process on the frontend, like any other program. Under the hood, the frontend process uses a cluster middleware library to start a backend process on each backend processor, load a copy of the program into each backend's memory, initialize connections for message passing through the backend network, and commence execution of each backend process. At program completion, when all the backend processes have terminated, the frontend process also terminates.

For maximum performance, it's not enough to equip a cluster with fast backend processors. It's also important for the cluster's backend network to have three characteristics: low latency, high bandwidth, and high bisection bandwidth.

**Latency** refers to the amount of time needed to start up a message, regardless of the message's size; it depends on the hardware and software protocols used on the network. **Bandwidth**, measured in bits per second, is the rate at which data is transmitted once a message has started. The time to transfer a message is the latency, plus the message size divided by the bandwidth. A small latency and a large bandwidth will minimize each message's transfer time, thus reducing the cluster parallel program's running time.

**Bisection bandwidth**, also measured in bits per second, refers to the total rate at which data can be transferred if half the nodes in the cluster are sending messages to the other half. In other words, the network is split down the middle—bisected—and each node on one side of the split sends data as fast as possible to a different node on the other side of the split. As we will see in Part III, some cluster parallel programs do in fact send messages from half the processes to the other half at the same time. Ideally, in an $N$-node cluster, the network's bisection bandwidth would be $N/2$ times the bandwidth on a single link. Depending on how the backend network is built, however, the bisection bandwidth may be less than the ideal, which in turn may reduce the cluster parallel program's performance.

Several commodity off-the-shelf technologies are used for backend networks in cluster parallel computers. The available alternatives fall into two categories: Ethernet, and everything else.

**Ethernet**, due to its ubiquity, is the least-expensive alternative. Ethernet interface cards, switches, and cables that support a bandwidth of 1 gigabit per second (1 Gbps), or $1 \times 10^9$ bits per second, are readily available. Although more expensive, 10 Gbps Ethernet equipment is also available, and 100 Gbps Ethernet is on the horizon. An Ethernet *switch* usually has a large bisection bandwidth. (An Ethernet *hub* does not; you should never use a hub to build a cluster backend network.) Platform-independent programs that communicate over Ethernet are easily written using the standard socket application program interface (API) and the Internet standard TCP/IP protocols. However, Ethernet has a much higher latency—around 150 microseconds (150 $\mu$sec), or $150 \times 10^{-6}$ seconds—than the alternatives, especially when using TCP/IP.

Other interconnection technologies used in cluster parallel computers, such as **InfiniBand**, **Scalable Coherent Interface (SCI)**, and **Myrinet**, are all more or less the same in their gross characteristics. They all support higher bandwidths than Ethernet (up to about 100 Gbps), much lower latencies than Ethernet (in the single microsecond range), and high bisection bandwidths. However, not being nearly as widespread as Ethernet, they are all more expensive. They also tend to require the use of technology-specific software libraries to achieve their full performance. While TCP/IP can be layered on top of these technologies, thus allowing platform-independent programs to run on these technologies, layering TCP/IP increases the latency due to the TCP/IP protocol overhead.

Ultra-high-performance parallel supercomputers often use interconnect technologies such as InfiniBand, SCI, or Myrinet because of their higher bandwidth and lower latency. Mundane parallel computers usually use Ethernet because of its lower cost and platform-independent programming support.

## 2.5 Hybrid Parallel Computers

A **hybrid** parallel computer is a cluster in which each backend processor is an SMP machine (Figure 2.10). In other words, it is a combination, or hybrid, of cluster and SMP parallel computers. A hybrid parallel computer has both shared memory (within each backend) and distributed memory (between backends). Eventually, all commodity clusters will be hybrid parallel computers because multicore PCs are becoming popular and single-core PCs are becoming harder to find.

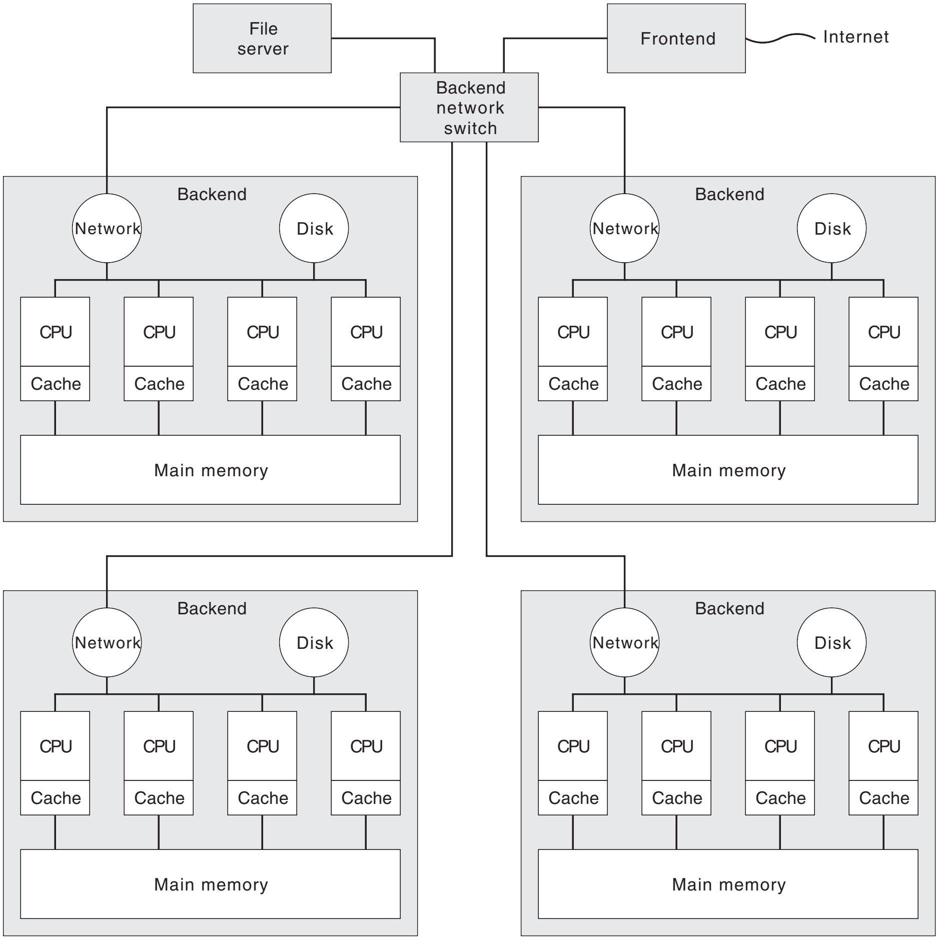


**Figure 2.10** Hybrid parallel computer

A hybrid parallel computer is programmed using a combination of cluster and SMP parallel programming techniques (Figure 2.11). Like a cluster parallel computer, each backend runs a separate process with its own address space. Each process executes a copy of the same program, and each process has a portion of the data. Like an SMP parallel computer, each process in turn has multiple threads, one thread running on each CPU. Threads in the same process share the same address space and can access their own shared data structures directly. Threads in different processes must send messages to each other to transfer data.

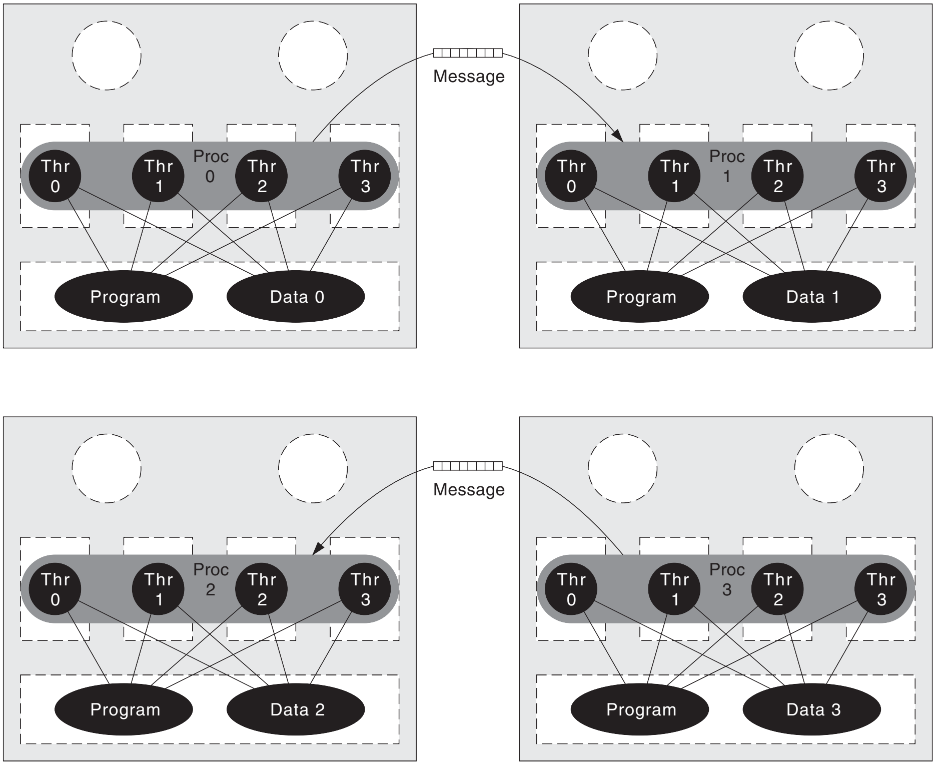


**Figure 2.11** Hybrid parallel program

## 2.6 Computing Grids

Architecturally, a **computing grid** is the same as a cluster parallel computer, except that the processors are not all located in the same place and are not all connected to a common, dedicated backend network. Instead, the processors are located at diverse sites and are connected through a combination of local area networks and the Internet (Figure 2.12).

Often, a grid is set up by a consortium of companies, universities, research institutions, and government agencies—the member organizations contributing computers to the grid. The Open Science Grid (OSG), for example, is a grid devoted to large-scale scientific computation, with thousands of processors located at 85 institutions in Brazil, Canada, England, Germany, Korea, Taiwan, and the United States.

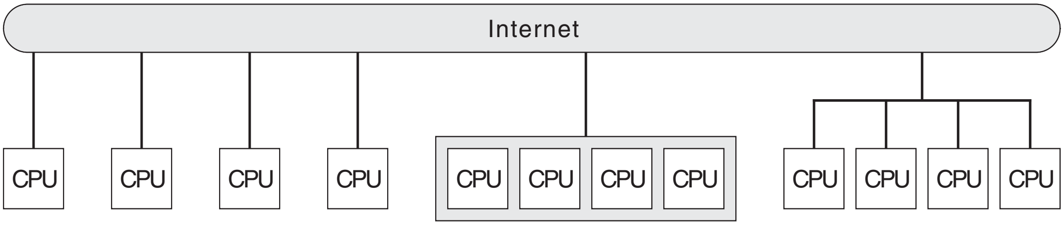


**Figure 2.12** Computing grid with single processors, an SMP, and a cluster

Other grids are based on "volunteer computing." Users download a special client program to their desktop PCs. The client program runs as a low-priority background process, or sometimes as a screen saver. When the PC becomes idle, the client program starts up and executes a portion of a parallel computation, communicating with other nodes over the Internet. Projects using volunteer computing grids include SETI@home, the Great Internet Mersenne Prime Search (GIMPS), and dozens of others. The Berkeley Open Infrastructure for Network Computing (BOINC) is a general framework for developing parallel programs that run on volunteer computing grids.

Computing grids are programmed in the same way as cluster parallel computers, with multiple processes running on the various grid machines. However, a parallel program that performs well on a cluster is not necessarily well suited for a grid. The Internet's latency is orders of magnitude larger, and the Internet's bandwidth is orders of magnitude smaller, than a typical cluster backend network. Because some of the computers running the grid program may be connected through the Internet, the average message takes a lot longer to send on a grid than on a cluster. Thus, parallel programs that require intensive message passing do not perform well on a grid. Problems best suited for a grid are those that can be divided into many pieces that are computed independently with little or no communication.

In this book, we will study how to build parallel programs for *tightly coupled* processors: SMP parallel computers where all processors use a single shared memory; and cluster and hybrid parallel computers where all processors are connected to the same high-speed backend network. Parallel programming for *loosely coupled* computing grids is beyond the scope of this book.

## 2.7 GPU Coprocessors

CPU, memory, and Ethernet chips are not the only chips that have become commodities. Driven by the market's insatiable appetite for ever-higher-performing graphics displays on PCs and game consoles, **graphics processing unit (GPU)** chips have also become commonplace.

Acting as a **coprocessor** to the main CPU, the GPU is a specialized chip that handles graphics rendering calculations at very high speeds. The CPU sends high-level commands to the GPU to draw lines and fill shapes with realistic shading and lighting; the GPU then calculates the color of each pixel and drives the display. Because the pixels can be computed independently, the GPU typically has multiple processing cores and calculates multiple pixels in parallel.

Programmers have realized that GPUs can be used to do parallel computations other than pixel rendering, and have even coined an acronym for it: **GPGPU**, or General Purpose computation on Graphics Processing Units. In response, GPU vendors have repackaged their graphics cards as general-purpose massively parallel coprocessors, complete with on-board shared memory and with APIs for parallel programming on the GPU. A GPU coprocessor card transforms a regular PC into what marketers call a "desktop supercomputer" (Figure 2.13). The low cost of the commodity GPU chips makes a GPU coprocessor an attractive alternative to general-purpose multicore CPUs and clusters.

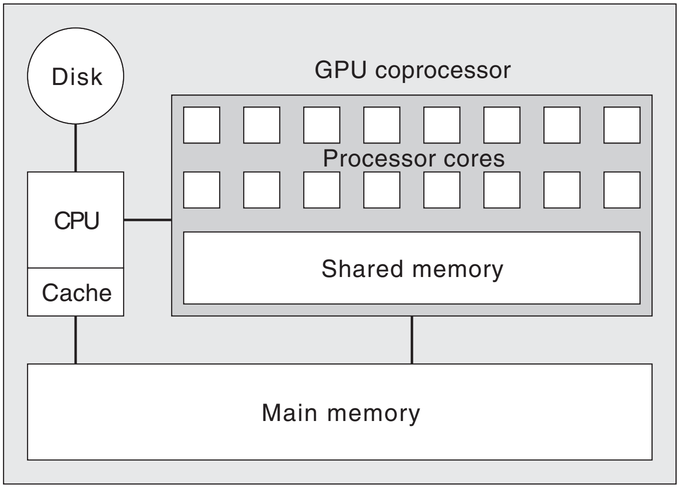


**Figure 2.13** Parallel computer with GPU coprocessor

A GPU's instruction set is usually more limited than a regular CPU's. For example, a GPU may support single-precision floating-point arithmetic, but not double precision. Also, the GPU's cores typically must all perform identical operations simultaneously, each core operating on its own data items. For these reasons, current GPU coprocessors cannot run arbitrary parallel programs. However, GPU coprocessors excel at running "inner loops," where the same statements are performed on every element of an array or matrix at very high speed in parallel. Thus, a GPU parallel program usually consists of regular code executed on the main CPU with a computation-intensive section executed on the GPU. Unfortunately, space limitations do not permit covering GPU parallel programming in this book.

## 2.8 SMPs, Clusters, Hybrids: Pros and Cons

Why are there three prevalent parallel computer architectures? Why not use the same architecture for all parallel computers? The reason is that each architecture is best suited for certain kinds of problems and is not well suited for other kinds of problems.

An SMP parallel computer is well suited for a problem where there are data dependencies between the processors—where each processor produces results that are used by some or all of the other processors. By putting the data in a common address space (shared memory), each thread can access every other thread's results directly at the full speed of the CPU and memory. A cluster parallel computer running such a program would have to send many messages between the processors. Because sending a message is orders of magnitude slower than accessing a shared memory location, even with high-speed interconnects such as InfiniBand, SCI, and Myrinet, an SMP parallel computer would outperform a cluster parallel computer on this problem.

Conversely, a cluster parallel computer is well suited for a problem where there are few or no data dependencies between the processors—where each processor can compute its own results with little or no communication with the other processors. The more communication needed, the poorer a cluster parallel program will perform compared to an SMP parallel program.

On the other hand, two limitations are encountered when trying to scale up to larger problem sizes on an SMP parallel computer. First, there is a limit on the physical memory size. A 32-bit CPU can access at most 4 gigabytes ($2^{32}$ bytes) of physical memory. While a 64-bit CPU can theoretically access up to 16 million terabytes ($2^{64}$ bytes) of physical memory, it will be a long time before any single CPU has a physical memory that large. Although virtual memory lets a process access a larger address space than physical memory, the performance of a program whose instructions and data do not fit in physical memory would be severely reduced due to swapping pages back and forth between physical memory and disk. Second, there is a limit on the number of CPUs that can share the same main memory. The more CPUs there are, the more circuitry is needed to coordinate the memory transactions from all the CPUs and to keep all the CPUs' caches coherent. Thus, an SMP parallel program's problem size cannot scale up past the point where its data no longer fits in main memory, and its speedup or sizeup cannot scale up past a certain number of CPUs.

With a cluster parallel computer, there are no limits on scalability due to main memory size or number of processors. On a *K*-node cluster, the maximum total amount of memory is *K* times the maximum on one node. Thus, if you need more memory, more speedup, or more sizeup, just add more nodes to the cluster. However, because message latency tends to increase as the number of nodes on the network increases, causing program performance to decrease, a cluster cannot keep growing forever. Still, commodity clusters with hundreds and even thousands of nodes have been built; the largest commodity SMPs have dozens of nodes at most.

Like a cluster, a hybrid parallel computer can be scaled up to larger problem sizes by adding more nodes. In addition, a hybrid parallel computer is especially well suited for a problem that can be broken into chunks having few or no data dependencies between chunks, but that can have significant data dependencies within each chunk. The chunks can be computed in parallel by separate processes running on the cluster nodes and passing messages among each other. Within each chunk, the results can be computed in parallel by separate threads running on the node's CPUs and accessing data in shared memory. Many of the supercomputers in the TOP500 List are commodity hybrid parallel computers.

Any parallel program can be implemented to run on an SMP, cluster, or hybrid parallel computer. (For some of the example programs in this book, we will look at all three variations.) Which kind of parallel computer, then, should you use to solve your high-performance computing problem—assuming your local computer center even offers you a choice? The answer is to use the kind of parallel computer that gives the best performance on your problem. However, because so many factors influence performance, the only way to know for sure is to implement the appropriate versions of the program and try them on the available parallel computers. As we study how to design and code parallel programs, we will also discuss how the program's design influences the program's performance, and we will see how certain kinds of parallel programs perform better on certain kinds of parallel computers.

## 2.9 Parallel Programming Libraries

It is perfectly possible to write parallel programs using a standard programming language and generic operating system kernel functions. You can write a multithreaded program in C using the standard POSIX thread library (Pthreads), or in Java using Java's built-in Thread class. If you run this program on an SMP parallel computer, each thread will run on a different processor simultaneously, yielding a parallel speedup. You can write a multiprocess program in C using the standard socket API to communicate between processes, or in Java using Java's built-in java.net.Socket class. If you run copies of this process on the backend processors of a cluster parallel computer, the simultaneously running processes will yield a parallel speedup.

However, parallel programs are generally not written this way, for two reasons. First, some programming languages popular in domains that benefit from parallel programming—such as Fortran for scientific computing—do not support multithreaded programming and network programming as well as other languages. Second, using low-level thread and socket libraries increases the effort needed to write a parallel program. It takes a great deal of effort to write the code that sets up and coordinates the multiple threads of an SMP parallel program, or to write the code that sets up network connections and sends and receives messages for a cluster parallel program. Parallel program users are interested in solving problems in their domains, such as searching a massive DNA sequence database or calculating the motion of stars in a galaxy, not in writing thread or network code. Indeed, many parallel program users may lack the expertise to write thread or network code.

Instead, to write a parallel program, you use a **parallel programming library**. The library encapsulates the low-level thread or network code that is the same in any parallel program, presenting you with easy-to-use, high-level parallel programming abstractions. You can then devote most of your parallel programming effort to solving your domain problem using those abstractions.

**OpenMP** is the standard library for SMP parallel programming. OpenMP supports the Fortran, C, and C++ languages. By inserting special OpenMP **pragmas** into the source code, you designate which sections of code are to be executed in parallel by multiple threads. You then run the annotated source code through a special OpenMP compiler. The OpenMP compiler looks at the OpenMP pragmas, *rewrites* your source code to add the necessary low-level threading code, and compiles your now-multithreaded program as a regular Fortran, C, or C++ program. You then run your program as usual on an SMP parallel computer. See Appendix A for further information about OpenMP.

The **Message Passing Interface (MPI)** is the standard library for cluster parallel programming. Like OpenMP, MPI supports the Fortran, C, and C++ languages. Unlike OpenMP, MPI requires no special compiler; it is just a subroutine library. You write your parallel program like any regular program, calling the MPI library routines as necessary to send and receive messages, and compile your program as usual. To run your program on a cluster parallel computer, you execute a special MPI **launcher** program. The launcher takes care of starting a process to run your compiled executable program on each backend processor. The MPI library routines your program calls then take care of all the details of setting up network connections between processes and passing messages back and forth. See Appendix B for further information about MPI.

As already mentioned, hybrid parallel computers are becoming popular, due to the wide availability of multicore PCs. However, as yet, there are no standard libraries for hybrid parallel programming. OpenMP has no routines for message passing. MPI has no capabilities for executing sections of code in parallel in multiple threads. Writing a hybrid parallel program using both OpenMP and MPI is not guaranteed to work, because the MPI standard does not require all MPI implementations to support multithreaded programs (although an MPI implementation is allowed to do so). While hybrid parallel programs can be written solely using MPI by running a separate *process* (rather than a thread) on each CPU of each node, sending messages between different processes' address spaces on the same node often yields poorer performance than simply sharing the same address space among several threads.

In addition, neither the official OpenMP standard nor the official MPI standard supports the Java language. Yet Java is becoming the language that most computing students learn. While several unofficial versions of MPI and OpenMP in Java have appeared, none can be considered a standard, and none are designed for hybrid parallel programming.

In this book, we will use the **Parallel Java Library** to learn how to build parallel programs. Parallel Java includes both the multithreaded parallel programming capabilities of OpenMP and the message-passing capabilities of MPI, integrated in a single library. Thus, Parallel Java is well suited for hybrid parallel programming as well as SMP and cluster parallel programming. Parallel Java also includes its own middleware for running parallel programs on a cluster. Because the library is written in 100% Java, Parallel Java programs are portable to any machine that supports Java (JDK 1.5). Appendices A and B compare and contrast Parallel Java with OpenMP and MPI.

## 2.10 For Further Information

On the history of parallel computers:

- G. Wilson. A chronology of major events in parallel computing. University of Toronto Computer Systems Research Institute Technical Report CSRI–312, December 1994. ftp://ftp.cs.toronto.edu/csrg-technical-reports/312/csri312.ps

On Beowulf:

- T. Sterling, D. Becker, D. Savarese, J. Dorband, U. Ranawake, and C. Packer. Beowulf: A parallel workstation for scientific computation. In *Proceedings of the 24th International Conference on Parallel Processing (ICPP 1995)*, 1995, volume 1, pages 11–14.
- D. Ridge, D. Becker, P. Merkey, and T. Sterling. Beowulf: harnessing the power of parallelism in a Pile-of-PCs. In *Proceedings of the 1997 IEEE Aerospace Conference*, 1997, volume 2, pages 79–91.
- Beowulf.org Web site. http://www.beowulf.org/

On the Stone SouperComputer (and the tale of "Stone Soup"):

- W. Hargrove, F. Hoffman, and T. Sterling. The do-it-yourself supercomputer. *Scientific American*, 265(2):72–79, August 2001.
- The Stone SouperComputer. http://www.extremelinux.info/stonesoup/

On the Parallel Virtual Machine (PVM) library:

- V. Sunderam. PVM: a framework for parallel distributed computing. *Concurrency Practice and Experience*, 2(4):315–339, December 1990.
- A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine: A Users' Guide and Tutorial for Networked Parallel Computing*. MIT Press, 1994.
- PVM: Parallel Virtual Machine.
  http://www.csm.ornl.gov/pvm/pvm_home.html

On the official MPI standard:

- Message Passing Interface Forum Web Site.
  http://www.mpi-forum.org/
- Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*. June 12, 1995. (MPI Version 1.1)
  http://www.mpi-forum.org/docs/mpi-11.ps
- Message Passing Interface Forum. *MPI-2: Extensions to the Message-Passing Interface*. July 18, 1997. (MPI Version 2.0)
  http://www.mpi-forum.org/docs/mpi-20.ps

A comparison of PVM and MPI:

- G. Geist, J. Kohl, and P. Papadopoulos. PVM and MPI: a comparison of features. *Calculateurs Paralleles*, 8(2):137–150, 1996.

On the official OpenMP standard:

- OpenMP.org Web Site. http://openmp.org/wp/
- OpenMP Architecture Review Board. *OpenMP Application Program Interface, Version 3.0*. May 2008.
  http://www.openmp.org/mp-documents/spec30.pdf

On the TOP500 supercomputer list and the LINPACK benchmark:

- TOP500 Supercomputer Sites. http://www.top500.org/
- LINPACK. http://www.netlib.org/linpack/
- LINPACK Benchmark—Java Version.
  http://www.netlib.org/benchmark/linpackjava/

On the Open Science Grid:

- Open Science Grid. http://www.opensciencegrid.org/

On volunteer computing grids:

- Distributed Computing Projects directory. http://www.distributedcomputing.info/
- SETI@home. http://setiathome.berkeley.edu/
- GIMPS. http://www.mersenne.org/
- Berkeley Open Infrastructure for Network Computing. http://boinc.berkeley.edu/

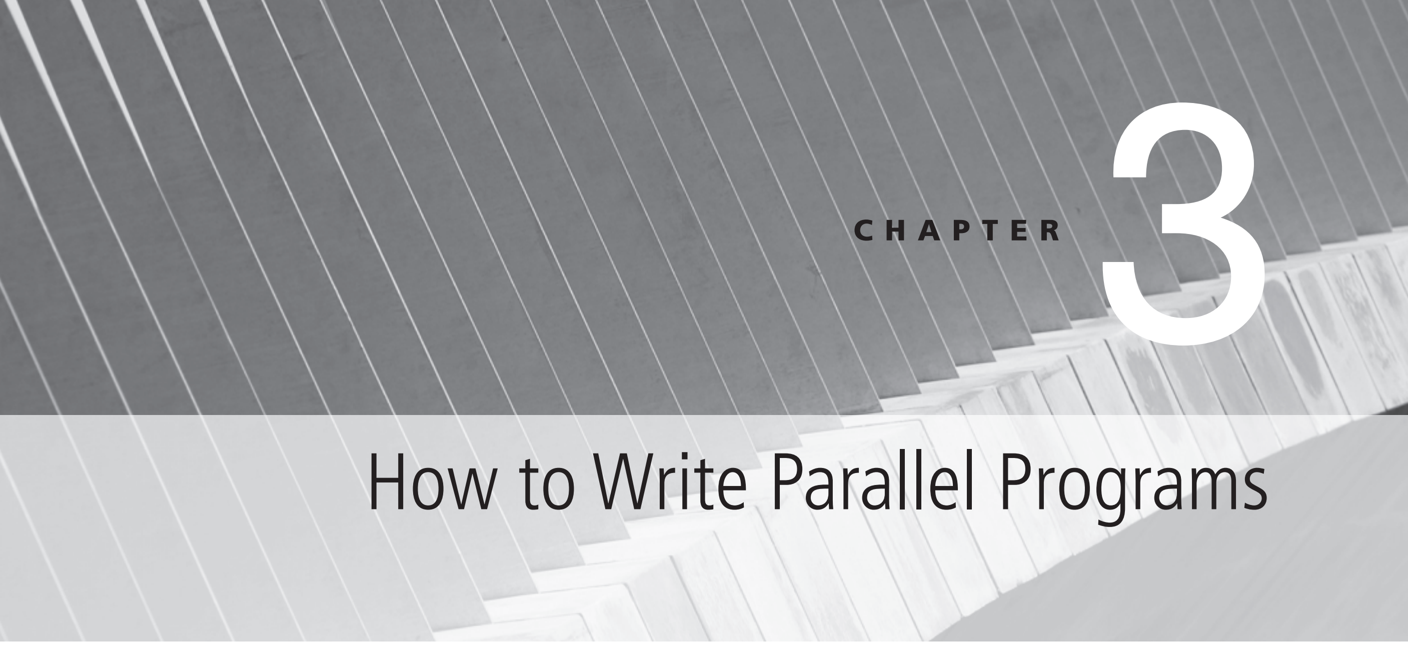
On parallel computing with GPUs:

- GPGPU Web Site. http://www.gpgpu.org/
- N. Goodnight, R. Wang, and G. Humphreys. Computation on programmable graphics hardware. *IEEE Computer Graphics and Applications*, 25(5):12–15, September/October 2005.
- J. Owens, M. Houston, D. Luebke, S. Green, J. Stone, and J. Phillips. GPU computing. *Proceedings of the IEEE*, 96(5):879–899, May 2008.
- R. Fernando, editor. *GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics*. Pearson Education, 2004.
- M. Pharr, editor. *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*. Pearson Education, 2005.
- H. Nguyen, editor. *GPU Gems 3*. Addison-Wesley, 2007.

On the Parallel Java Library:

- A. Kaminsky. Parallel Java: a unified API for shared memory and cluster parallel programming in 100% Java. In *Proceedings of the 21st IEEE International Parallel and Distributed Processing Symposium (IPDPS 2007),* March 2007.
- Parallel Java Library (including downloads). http://www.cs.rit.edu/~ark/pj.shtml
- Parallel Java documentation. http://www.cs.rit.edu/~ark/pj/doc/index.html

*This page intentionally left blank*



# CHAPTER 3

# How to Write Parallel Programs

in which we discover the three principal patterns for designing parallel programs; we encounter examples of problems for which each pattern is suited; and we see how to realize the patterns on practical parallel computers

# CHAPTER 3 How to Write Parallel Programs

## 3.1 Patterns of Parallelism

Let's say you are given a problem that requires a massive amount of computation—such as finding, in an enormous genomic database of DNA sequences, the few sequences that best match a short query sequence; or calculating the three-dimensional positions as a function of time of a thousand stars in a star cluster under the influence of their mutual gravitational forces. To get the answer in an acceptable amount of time, you need to write a parallel program to solve the problem. But where do you start? How do you even think about designing a parallel program?

In an 1989 paper titled "How to write parallel programs: a guide to the perplexed," Nicholas Carriero and David Gelernter of Yale University addressed this question by codifying three patterns for designing parallel programs: **result parallelism**; **agenda parallelism**; and **specialist parallelism**. Each pattern encompasses a whole class of similarly structured problems; furthermore, each pattern suggests how to design a parallel program for any problem in that class. Using the patterns, the steps for designing a parallel program are the following:

- Identify the pattern that best matches the problem.
- Take the pattern's suggested design as the starting point.
- Implement the design using the appropriate constructs in a parallel programming language.

Next, we describe each of the three patterns and give examples of how they are used.

## 3.2 Result Parallelism

In a problem that exhibits **result parallelism** (Figure 3.1), there is a collection of multiple results. The individual results are all computed in parallel, each by its own processor. Each processor is able to carry out the complete computation to produce one result. The conceptual parallel program design is:

```
Processor 1:    Compute result 1
Processor 2:    Compute result 2
. . .
Processor N:    Compute result N
```

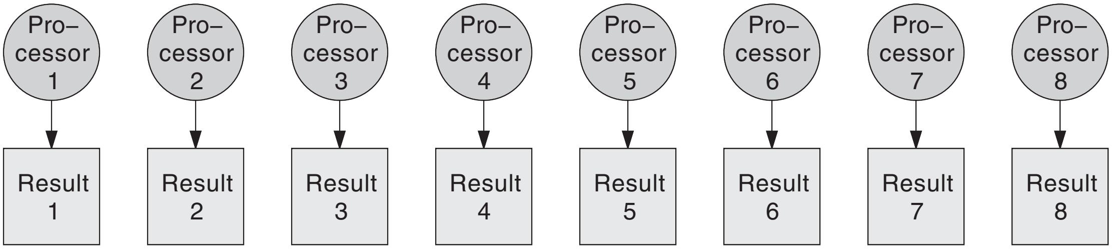


**Figure 3.1** Result parallelism

A problem that requires computing each element of a data structure often exhibits result parallelism. As an example, consider the problem of calculating all the pixels in all the frames of a computer-animated film. One way to solve the problem is to assign a separate processor to calculate each pixel. Another way is to assign a separate processor to render each entire frame, calculating all the pixels in that frame. The former is an example of **fine-grained parallelism**, where each result requires a small amount of computation. The latter is an example of **coarse-grained parallelism**, where each result requires a large amount of computation. In both cases, though, none of the processors needs to use the result calculated by any other processor; there are no dependencies among the computations. Thus, in concept, all the processors can start at the same time, calculate, and finish at the same time.

Other problems, however, do have dependencies among the computations. Consider calculating the 3-D positions of $N$ stars in a star cluster as a function of time for a series of $M$ time steps. The result can be viewed as an $M \times N$-element matrix of positions: row 1 contains the stars' positions after the first time step; row 2 contains the stars' positions after the second time step; and so on. In concept, the parallel program has $M \times N$ processors. The processors in row 1 can begin computing their results immediately. Each processor in row 1 calculates the gravitational force on its star due to each of the other stars, using the stars' input initial positions. Each processor in row 1 then moves its star by one time step in a direction determined by the net gravitational force, and the star's new position becomes the processor's result. However, the processors in row 2 cannot begin computing their results immediately. To do their computations, these processors need the stars' positions after the first time step, so these processors must wait until all the row-1 processors have computed their results. We say there are **sequential dependencies** from the computations in each row to the computations in the next row (Figure 3.2). There are no sequential dependencies between the computations in the same row, though.

Faced with the problem of calculating stellar motion for, say, one thousand stars and one million time steps, you might wonder where to find a parallel computer with enough hardware to compute each element of the result in its own separate processor. Keep in mind that, for now, we are still in the realm of *conceptual* parallel program design, where there are no pesky hardware limitations. In Section 3.5 we will see how to translate this conceptual design into a real parallel program.

Recalculating a spreadsheet is another example of a result parallel problem with sequential dependencies. The spreadsheet cell values are the results computed by the program. Conceptually, each cell has its own processor that computes the value of the cell's formula. When you type a new value into an input cell, the processors all calculate their respective cells' formulas. Normally, all the cells can be calculated in parallel. However, if the formula for cell B1 uses the value of cell A1, then the B1 processor must wait until the A1 processor has finished. Soon all spreadsheets will have to be result parallel programs to get full performance out of a desktop or laptop computer's multicore CPU chip.

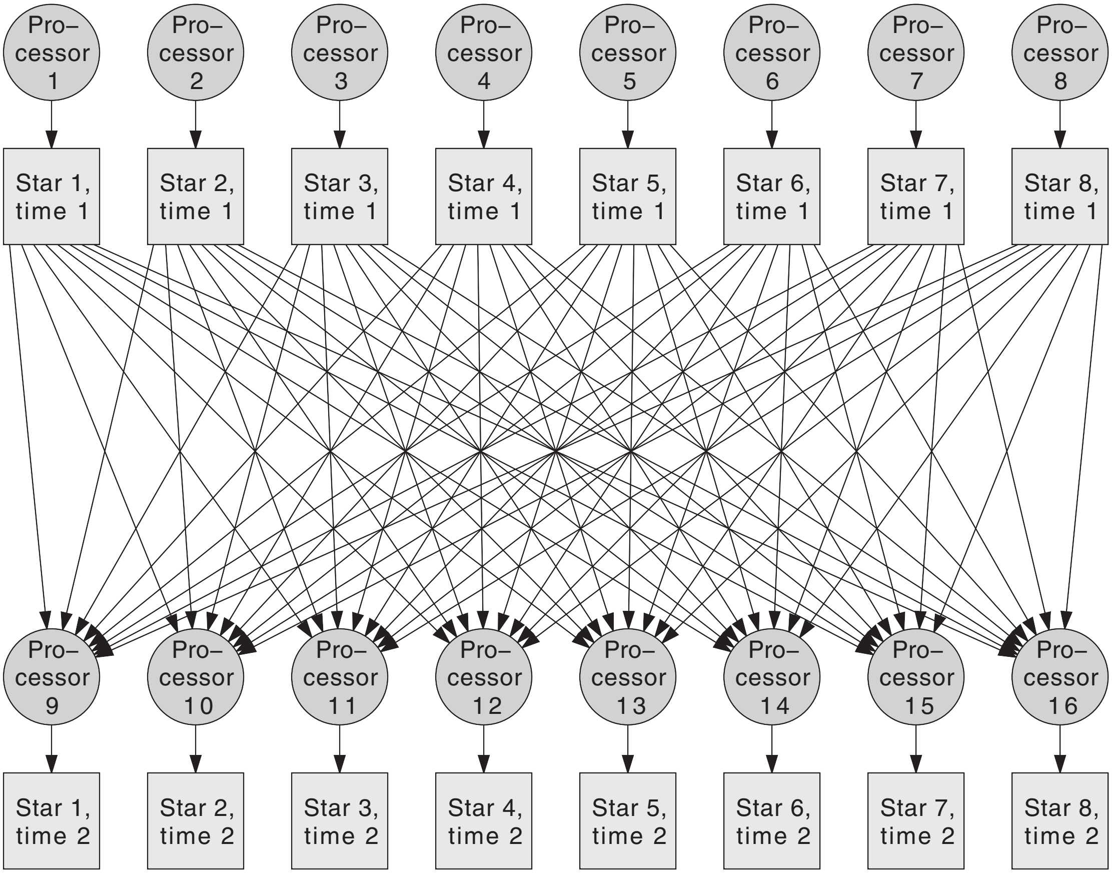


**Figure 3.2** Result parallelism with sequential dependencies—star cluster simulation

## 3.3 Agenda Parallelism

In a problem that exhibits **agenda parallelism** (Figure 3.3), there is an agenda of tasks that must be performed to solve the problem, and there is a team of processors, each processor able to perform any task. The conceptual parallel program design is:

Processor 1: Perform task 1

Processor 2: Perform task 2

. . .

Processor *N*: Perform task *N*

A problem that requires computing one result, or a small number of results, from a large number of inputs often exhibits agenda parallelism. Querying a DNA sequence database is one example. The agenda items are: “Determine if the query sequence matches database sequence 1”; “Determine if the query sequence matches database sequence 2”; and so on. Each of these tasks can be performed independently of all the others, in parallel.

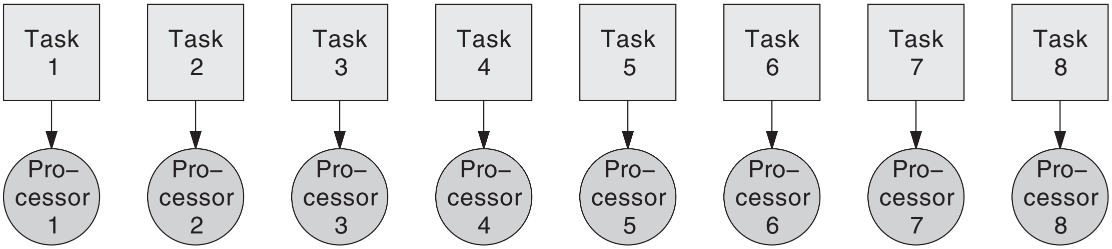


**Figure 3.3** Agenda parallelism

Other agenda parallel problems have sequential dependencies among the tasks (Figure 3.4). Certain tasks cannot start until other tasks have finished. The Basic Local Alignment Search Tool (BLAST) program, a widely used DNA and protein sequence database search program, can be viewed as an agenda parallel problem. BLAST proceeds in a series of phases. In the first phase, BLAST looks for matches between short pieces of the query sequence and short pieces of the sequence database; this results in a large number of tentative starting points known as “seeds.” In the second phase, BLAST takes each seed and tries to align the complete query sequence with the sequence database, starting from the seed’s location. BLAST computes a score for each alignment that tells how biologically plausible the alignment is; alignments that don’t result in a good match (too low a score) are discarded. In the third phase, BLAST sorts the surviving alignments into descending order of plausibility and outputs the alignments, most plausible first. Conceptually, the phase-1 agenda items are of the form “For seed *X*, match piece *Y* of the query against piece *Z* of the database.” These can all be done in parallel. The phase-2 agenda items are of the form “Align the query with the database at seed *X*’s location and compute the plausibility score.” These can all be done in parallel with each other, but each must wait until the corresponding phase-1 agenda item has finished. The final agenda item, “Sort and output the alignments,” must wait until the phase-2 agenda items have finished.

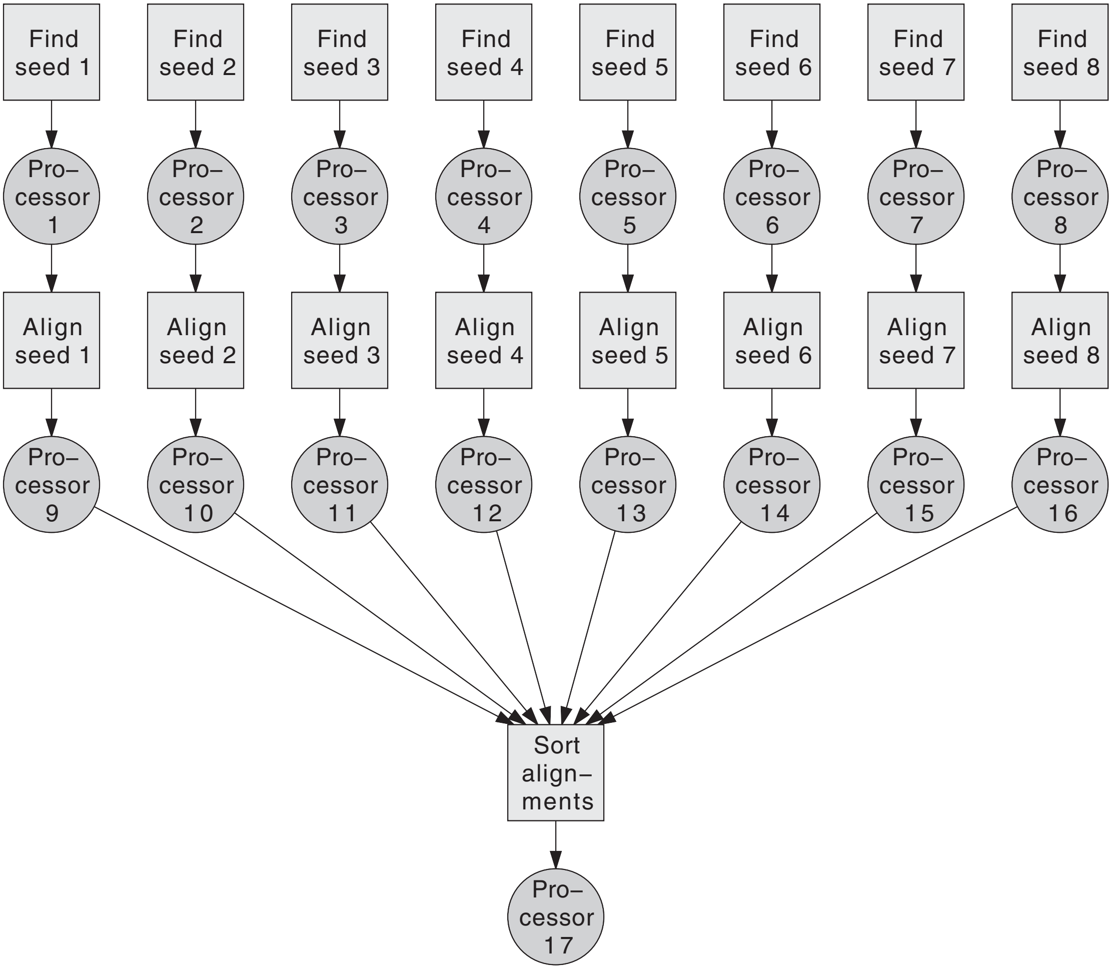


**Figure 3.4** Agenda parallelism with sequential dependencies—BLAST

A result parallel problem could be viewed as an agenda parallel problem, where the agenda items are "Compute result 1," "Compute result 2," and so on. The difference is that in a result parallel problem, we are typically interested in *every* processor's result. In an agenda parallel problem, we are typically not interested in every processor's (agenda item's) result, but only in certain results, or only in a combination or summary of the individual results.

When an agenda parallel program's output is a combination or summary of the individual tasks' results, the program is following the so-called **reduction** pattern (Figure 3.5). The number of results is *reduced* from many down to one. Often, the final result is computed by applying a **reduction operator** to the individual results. For example, when the operator is addition, the final result is the sum of the tasks' results. When the operator is minimum, the final result is the smallest of the tasks' results.

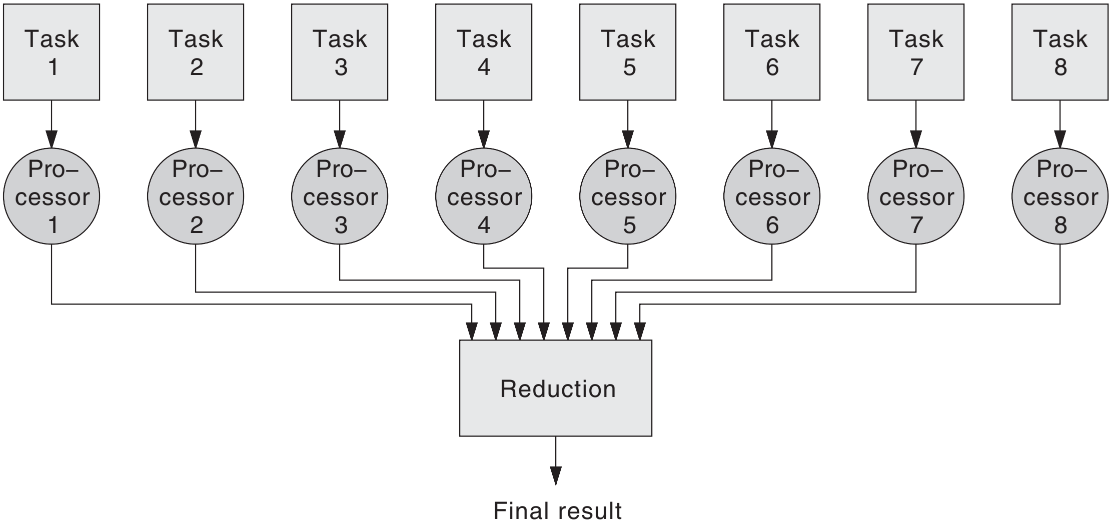


**Figure 3.5** Agenda parallelism with reduction

## 3.4 Specialist Parallelism

In a problem that exhibits **specialist parallelism** (Figure 3.6), like agenda parallelism, there is a group of tasks that must be performed to solve the problem, and there is a team of processors. But, unlike agenda parallelism, each processor performs only a specific one of the tasks, not just any task. Often, one specialist processor's job is to perform the same task on a series of items. The conceptual parallel program design is:

```
Processor 1:    For each item:
                    Perform task 1 on the item
Processor 2:    For each item:
                    Perform task 2 on the item
. . .
Processor $N$:  For each item:
                    Perform task $N$ on the item
```

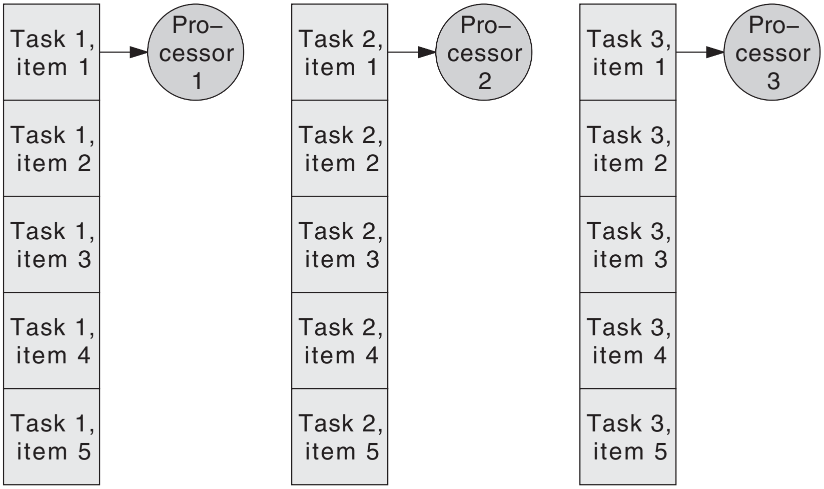


**Figure 3.6** Specialist parallelism

When there are sequential dependencies between the tasks in a specialist parallel problem, the program follows the so-called **pipeline** pattern. The output of one processor becomes the input for the next processor. All the processors execute in parallel, each taking its input from the preceding processor's previous output.

Consider again the problem of calculating the 3-D positions of *N* stars in a star cluster as a function of time for a series of *M* time steps. Now add a feature: At each time step, the program must create an image of the stars' positions and store the image in a Portable Network Graphics (PNG) file. This problem can be broken into three steps: calculate the stars' positions; create an image of the star's positions; and store the image in a PNG file. Each step requires a certain amount of computation: to calculate the numerical $(x,y,z)$ coordinates of each star; to determine the color of each pixel so as to display the stars' 3-D positions properly in the 2-D image; and to compress the pixel data and store it in a PNG file. The three steps can be performed in parallel by three processors in a specialist parallel program (Figure 3.7). While one processor is calculating the stars' positions for time step *t*, another processor is taking the stars' positions for time step *t*–1 and rendering an image, and a third processor is taking the image for time step *t*–2, compressing it, and storing it in a file. A program like this, where some processors are doing computations and other processors are doing file input or output, is said to be using the **overlapping** pattern, also called the **overlapped computation and I/O** pattern.

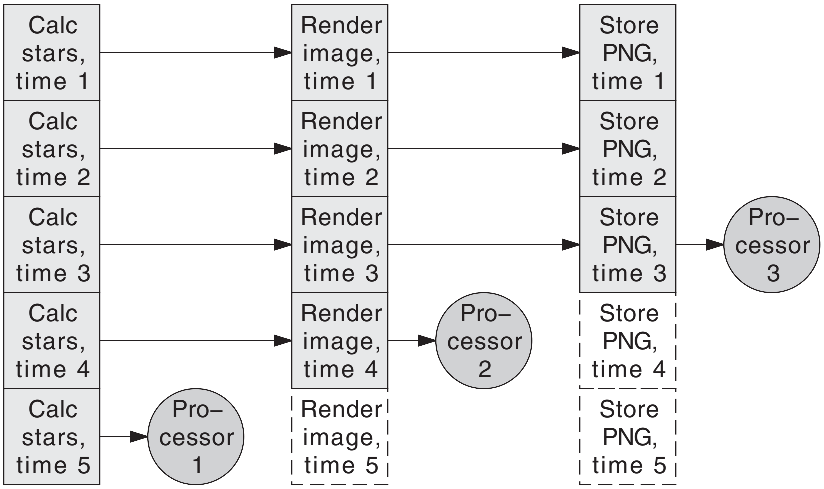


**Figure 3.7** Specialist parallelism with sequential dependencies—star cluster simulation

It's possible for a problem to exhibit multiple patterns of parallelism. The star cluster program, for example, can combine result parallelism with specialist parallelism. At each time step, we can have *N* processors each calculating one star's position (result parallelism); one processor rendering the image for the previous time step (specialist parallelism); and one processor compressing and writing the previous image to a PNG file (specialist parallelism). Putting it another way, the specialist parallel task of computing the stars' positions for one time step is itself a subproblem that exhibits result parallelism.

To sum up the three patterns: Result parallelism focuses on the results that can be computed in parallel. Agenda parallelism focuses on the tasks that can be performed in parallel. Specialist parallelism focuses on the processors that can execute in parallel.

## 3.5 Clumping, or Slicing

Applying the parallel program design patterns, as described so far, to a problem large enough to need a parallel computer would require a veritable horde of processors. A result parallel problem with a billion results would require a billion processors, one to compute each result. An agenda parallel problem with a billion agenda items would require a billion processors as well. (Specialist parallel problems tend not to require such large numbers of *different* specialists.) A problem size of one billion—$10^9$, or about $2^{30}$—is by no means far-fetched. We will run even the simple, pedagogical parallel programs in this book on problems of this size. Real-world parallel programs regularly run on much larger problems.

The difficulty, of course, is finding a parallel computer with billions and billions of processors. Well-funded government or academic high-performance computing centers may have parallel computers with processors numbering in the thousands. Most of us would count ourselves lucky to have a dozen or two.

To fit a large parallel problem on an actual parallel computer with comparatively few processors, we use **clumping**. Many conceptual processors are clumped together and executed by one actual processor. In a result parallel program, each processor computes a clump of many results instead of just one result (Figure 3.8).

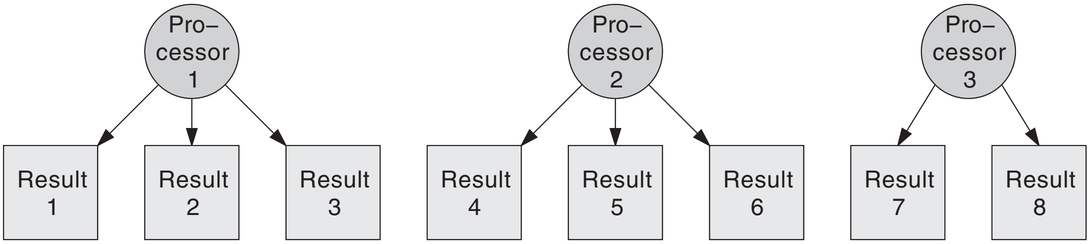


**Figure 3.8** Result parallelism with clumping (or slicing)

**Slicing** is another way of looking at the same thing. Rather than thinking of clumping many processors into one, think of dividing the result data structure into slices, as many slices as there are processors, and assigning one processor to compute all the results in the corresponding slice. For example, suppose there are 100 results and 4 processors. The design of the result parallel program with slicing is:

| | |
|---|---|
| Processor 1: | Compute result 1, 2, . . . 24, 25 |
| Processor 2: | Compute result 26, 27, . . . 49, 50 |
| Processor 3: | Compute result 51, 52, . . . 74, 75 |
| Processor 4: | Compute result 76, 77, . . . 99, 100 |

In the rest of the book, we will study several parallel programming constructs that automatically slice up a problem of any size to use however many processors the parallel computer has.

## 3.6 Master-Worker

An agenda parallel problem with many more tasks than processors must also use clumping on a real parallel computer (Figure 3.9). Each processor performs many tasks, not just one. Conceptually, the agenda takes the form of a bag of tasks. Each processor repeatedly takes a task out of the bag and performs the task, until the bag is empty, as follows:

```
Processor 1:    While there are more tasks:
                    Get and perform the next task
Processor 2:    While there are more tasks:
                    Get and perform the next task
                    . . .
Processor K:    While there are more tasks:
                    Get and perform the next task
```

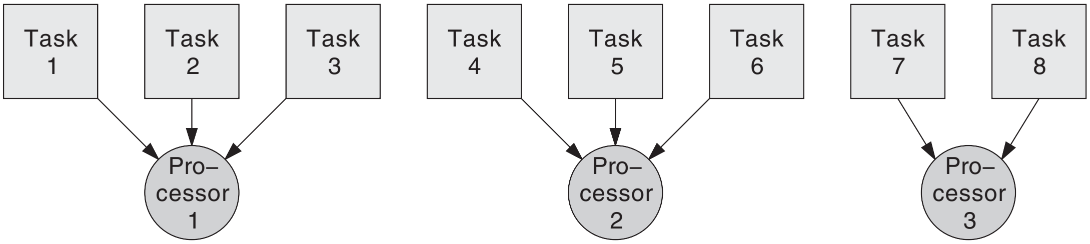


**Figure 3.9** Agenda parallelism with clumping

On a cluster parallel computer, an agenda parallel problem with clumping is often realized concretely using the **master-worker** pattern (Figure 3.10). There is one master processor in charge of the agenda, and there are *K* worker processors that carry out the agenda items. The master sends tasks to the workers, receives the task results from the workers, and keeps track of the program's overall results. Each worker receives tasks from the master, computes the task results, and sends the results back to the master. The conceptual parallel program design is:

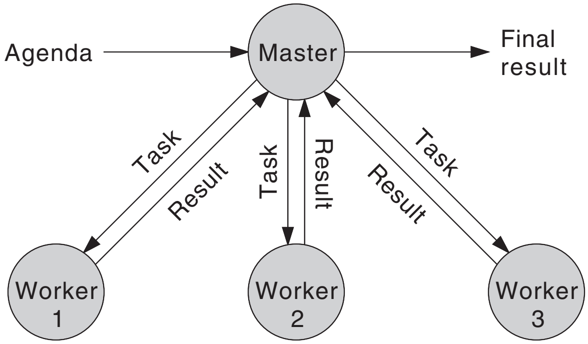


**Figure 3.10** Agenda parallelism, master-worker pattern

```
Master:      Send initial task to each worker
             Repeat:
                 Receive task result from any worker X
                 Record task result
                 Get next task
                 If there are no more tasks, tell worker X to stop
                 Otherwise, send task to worker X
Worker 1:    Repeat:
                 Receive a task from the master
                 If there are no more tasks, stop
                 Compute task results
                 Send results to the master
Worker 2:    Repeat:
                 Receive a task from the master
                 If there are no more tasks, stop
                 Compute task results
                 Send results to the master
. . .
Worker K:    Repeat:
                 Receive a task from the master
                 If there are no more tasks, stop
                 Compute task results
                 Send results to the master
```

In the rest of the book, we will study many parallel programs designed to run on SMP parallel computers, cluster parallel computers, and hybrid cluster parallel computers. All these programs, however, will follow one of the three parallel design patterns—result parallelism, agenda parallelism, specialist parallelism—or a combination thereof. Before diving into an in-depth study of the parallel programming constructs that let us implement these patterns, in Chapter 4 we will wet our toes with a small introductory parallel program.

## 3.7 For Further Information

On the three parallel design patterns—Carriero's and Gelernter's paper:

- N. Carriero and D. Gelernter. How to write parallel programs: a guide to the perplexed. *ACM Computing Surveys*, 21(3):323–357, September 1989.

Carriero and Gelernter later expanded their paper into a book:

- N. Carriero and D. Gelernter. *How to Write Parallel Programs: A First Course*. MIT Press, 1990.

A more recent book about parallel design patterns, coming from the "patterns movement" in software design:

- T. Mattson, B. Sanders, and B. Massingill. *Patterns for Parallel Programming*. Addison-Wesley, 2005.

On the Basic Local Alignment Search Tool (BLAST)—the original paper:

- S. Altschul, W. Gish, W. Miller, E. Myers, and D. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215(3):403–410, October 5, 1990.

Sequential implementations of BLAST:

- FSA-BLAST. http://www.fsa-blast.org/
- NCBI BLAST. http://www.ncbi.nlm.nih.gov/
- WU-BLAST. http://blast.wustl.edu/

Parallel implementations of BLAST:

- mpiBLAST. http://www.mpiblast.org/
- ScalaBLAST. http://hpc.pnl.gov/projects/scalablast/

# CHAPTER 4

# A First Parallel Program

in which we build a simple sequential program; we convert it to a program for an SMP parallel computer; we see how long it takes to run each version; and we get some insight into how parallel programs execute

# CHAPTER 4 A First Parallel Program

## 4.1 Sequential Program

To demonstrate a program that can benefit from running on a parallel computer, let's invent a simple computation that will take a long time. Here is a Java subroutine that decides whether a number *x* is prime using the **trial division** algorithm. The subroutine tries to divide *x* by 2 and by every odd number *p* from 3 up to the square root of *x*. If any remainder is 0, then *p* is a factor of *x* and *x* is not prime; otherwise *x* is prime. While trial division is by no means the fastest way to test primality, it suffices for this demonstration program.

```
private static boolean isPrime
   (long x)
   {
   if (x % 2 == 0) return false;
   long p = 3;
   long psqr = p*p;
   while (psqr <= x)
      {
      if (x % p == 0) return false;
      p += 2;
      psqr = p*p;
      }
   return true;
   }
```

Here is a main program that uses a loop to call the subroutine with the values of *x* specified on the command line.

```
static int n;
static long[] x;

public static void main
   (String[] args)
   throws Exception
   {
   n = args.length;
   x = new long [n];
   for (int i = 0; i < n; ++ i)
      {
      x[i] = Long.parseLong (args[i]);
      }
   for (int i = 0; i < n; ++ i)
      {
      isPrime (x[i]);
      }
   }
```

When we run the primality testing program, we want to know the time when each subroutine call starts and the time when each subroutine call finishes, relative to the time the program started. This tells us how long it took to run the subroutine. To measure these times, we use Java’s `System.currentTimeMillis()` method, which returns the wall clock time in milliseconds (msec). We record each instant in a variable, and postpone printing the results, so as to disturb the timing as little as possible while the program is running. It can take several msec to call `println()`, and we don’t want to include that time in our measurements.

```
static int n;
static long[] x;
static long t1, t2[], t3[];

public static void main
   (String[] args)
   throws Exception
   {
   t1 = System.currentTimeMillis();
   n = args.length;
   x = new long [n];
   for (int i = 0; i < n; ++ i)
      {
      x[i] = Long.parseLong (args[i]);
      }
   t2 = new long [n];
   t3 = new long [n];
   for (int i = 0; i < n; ++ i)
      {
      t2[i] = System.currentTimeMillis();
      isPrime (x[i]);
      t3[i] = System.currentTimeMillis();
      }
   }
```

Here is the complete Java class, Program1Seq, including code to print the running time measurements.

```
public class Program1Seq
    {
    static int n;
    static long[] x;
    static long t1, t2[], t3[];

 public static void main
    (String[] args)
    throws Exception
    {
    t1 = System.currentTimeMillis();
    n = args.length;
    x = new long [n];
    for (int i = 0; i < n; ++ i)
       {
       x[i] = Long.parseLong (args[i]);
       }
    t2 = new long [n];
    t3 = new long [n];
    for (int i = 0; i < n; ++ i)
       {
       t2[i] = System.currentTimeMillis();
       isPrime (x[i]);
       t3[i] = System.currentTimeMillis();
       }
    for (int i = 0; i < n; ++ i)
       {
       System.out.println
          ("i = "+i+" call start = "+(t2[i]-t1)+" msec");
       System.out.println
          ("i = "+i+" call finish = "+(t3[i]-t1)+" msec");
       }
    }

 private static boolean isPrime
    (long x)
    {
    if (x % 2 == 0) return false;
    long p = 3;
    long psqr = p*p;
    while (psqr <= x)
```

```
            {
            if (x % p == 0) return false;
            p += 2;
            psqr = p*p;
            }
         return true;
         }
      }
```

## 4.2 Running the Sequential Program

When run on an SMP parallel computer, Program1Seq prints the following. The parallel computer has four processors, each a 450 MHz Sun Microsystems UltraSPARC-II CPU, and 2 GB of shared main memory. (Running the program on a different computer would, in general, yield different results.) All four arguments happen to be prime numbers.

```
$ java Program1Seq 1000000000000037 1000000000000091 \
  1000000000000159 1000000000000187
i = 0 call start = 1 msec
i = 0 call finish = 3842 msec
i = 1 call start = 3842 msec
i = 1 call finish = 7663 msec
i = 2 call start = 7663 msec
i = 2 call finish = 11502 msec
i = 3 call start = 11502 msec
i = 3 call finish = 15342 msec
```

Plotting each subroutine call's start and finish on a timeline reveals how the program executes (Figure 4.1). The program executes each subroutine call in its entirety before going on to the next subroutine call. Because the program's statements are executed in sequence with no overlap in time, we call it a **sequential program**. There is no parallelism, even when running on a parallel computer.

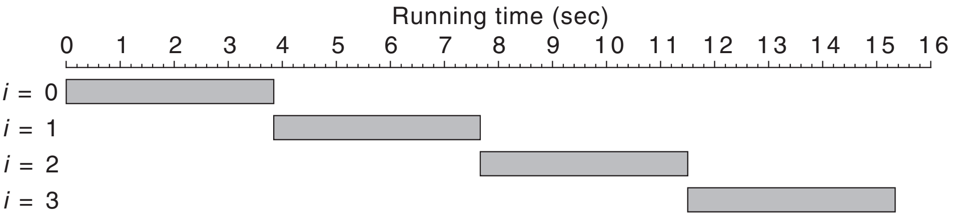


**Figure 4.1** Program1Seq execution timeline, SMP parallel computer

## 4.3 SMP Parallel Program

Now let's rewrite the program using Parallel Java so it will run in parallel when executed on an SMP parallel computer. In the main program, after extracting the command line arguments, we create a **parallel team** object. The constructor argument, `n`, says we want as many threads in the parallel team as there are values to test for primality.

```
public static void main
   (String[] args)
   throws Exception
   {
   n = args.length;
   x = new long [n];
   for (int i = 0; i < n; ++ i)
      {
      x[i] = Long.parseLong (args[i]);
      }
   new ParallelTeam(n);
   }
```

Each thread in the parallel team simultaneously executes the code in a **parallel region** object, declared here as an anonymous inner class. The actual parallel code goes in the parallel region's `run()` method.

```
public static void main
   (String[] args)
   throws Exception
   {
   n = args.length;
   x = new long [n];
   for (int i = 0; i < n; ++ i)
      {
      x[i] = Long.parseLong (args[i]);
      }
   new ParallelTeam(n).execute (new ParallelRegion()
      {
      public void run()
         {
         }
      });
   }
```

Rather than use a loop to execute the computations (subroutine calls) in sequence, we want the threads to execute the computations in parallel. To make this happen, we put the code for one computation in the parallel region's `run()` method. However, we want each computation to use a different *x* value. To make this happen, we set `i` to the index of the calling thread within the parallel team (0 through 3, as returned by the parallel region's `getThreadIndex()` method).

```
 public static void main
    (String[] args)
    throws Exception
    {
    n = args.length;
    x = new long [n];
    for (int i = 0; i < n; ++ i)
       {
       x[i] = Long.parseLong (args[i]);
       }
    new ParallelTeam(n).execute (new ParallelRegion()
       {
       public void run()
          {
          int i = getThreadIndex();
          isPrime (x[i]);
          }
       });
    }
```

Here is the complete Java class, Program1Smp, including code to record the running time measurements and print them after the parallel region has finished executing.

```
import edu.rit.pj.ParallelRegion;
import edu.rit.pj.ParallelTeam;
public class Program1Smp
    {
    static int n;
    static long[] x;
    static long t1, t2[], t3[];

 public static void main
    (String[] args)
    throws Exception
```

```
      {
      t1 = System.currentTimeMillis();
      n = args.length;
      x = new long [n];
      for (int i = 0; i < n; ++ i)
         {

         x[i] = Long.parseLong (args[i]);
         }
      t2 = new long [n];
      t3 = new long [n];
      new ParallelTeam(n).execute (new ParallelRegion()
         {
         public void run()
            {
            int i = getThreadIndex();
            t2[i] = System.currentTimeMillis();
            isPrime (x[i]);
            t3[i] = System.currentTimeMillis();
            }
         });
      for (int i = 0; i < n; ++ i)
         {
         System.out.println
            ("i = "+i+" call start = "+(t2[i]-t1)+" msec");
         System.out.println
            ("i = "+i+" call finish = "+(t3[i]-t1)+" msec");
         }
      }

   private static boolean isPrime
      (long x)
      {
      if (x % 2 == 0) return false;
      long p = 3;
      long psqr = p*p;
      while (psqr <= x)
         {
         if (x % p == 0) return false;
         p += 2;
         psqr = p*p;
         }
      return true;
      }
   }
```

Here's how the program works (Figure 4.2). The main program begins with one thread, the "main thread," executing the `main()` method. When the main thread creates the parallel team object, the parallel team object creates additional hidden threads; the constructor argument specifies the number of threads. These form a "team" of threads for executing code in parallel. When the main thread calls the parallel region's `execute()` method, the main thread suspends execution and the parallel team threads take over. All the team threads call the parallel region's `run()` method simultaneously, each thread retrieves a value for *x*, and each thread calls `isPrime()`. Thus, the `isPrime()` subroutine calls happen at the same time, and each subroutine call is performed by a different thread with a different argument. When all the subroutine calls have finished executing, the main thread resumes executing statements after the parallel region and prints the timing measurements.

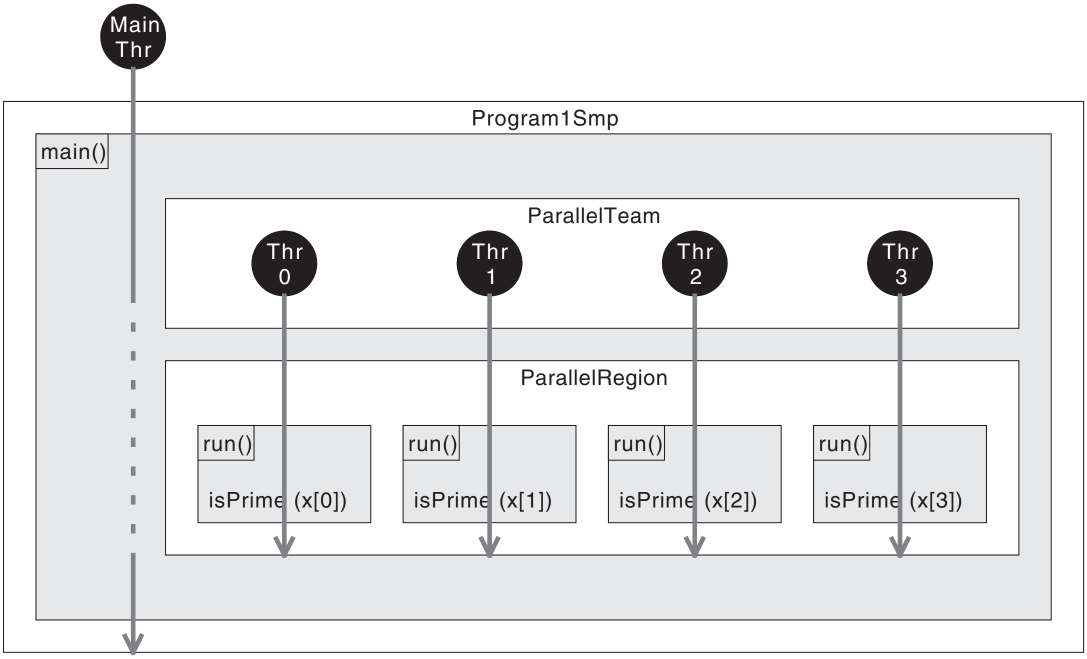


**Figure 4.2** Program1Smp operation

When running such a thread-based program on an SMP parallel computer, the Java Virtual Machine (JVM) and the operating system are responsible for scheduling each thread to execute on a different processor. Thus, the computations done by each thread—in this case, the different subroutine calls—are executed in parallel on different processors, resulting in a speedup with respect to the sequential program.

The parallel program illustrates a central theme of parallel program design: *Repetition does not necessarily imply sequencing.* The sequential program used a loop to get *n* repetitions of a subroutine call. As a side effect, the loop did the repetitions *in sequence.* However, for this program there is no need to do the repetitions in sequence. We wrote the original program with a loop because Java, like many programming languages, only has constructs for expressing a *sequence* of repetitions (a loop). So accustomed are we to this feature that whenever we are confronted with a repeated calculation, we automatically think "loop." However, a loop is not the only way to do a repeated calculation. Provided the repetitions do not have to be done in sequence, another way to do a repeated calculation is to run several copies of the calculation in multiple threads. A large part of the effort in learning parallel program design is breaking the habit of always using a loop to do repetitions in sequence, and forming the new habit of doing repetitions in parallel whenever possible.

## 4.4 Running the Parallel Program

When run on the four-processor parallel computer, Program1Smp printed the following:

```
$ java Program1Smp 1000000000000037 1000000000000091 \
  1000000000000159 1000000000000187
i = 0 call start = 125 msec
i = 0 call finish = 4076 msec
i = 1 call start = 125 msec
i = 1 call finish = 4098 msec
i = 2 call start = 125 msec
i = 2 call finish = 4082 msec
i = 3 call start = 125 msec
i = 3 call finish = 4076 msec
```

Now the timeline (Figure 4.3) shows parallelism. (Compare Figures 4.1 and 4.3 to Figure 1.2.) All the computations start at the same time, execute simultaneously, and finish at about the same time. Whereas the sequential version's running time was 15342 msec, the parallel version's running time on four processors was 4098 msec—a reduction by a factor of about four, as expected.

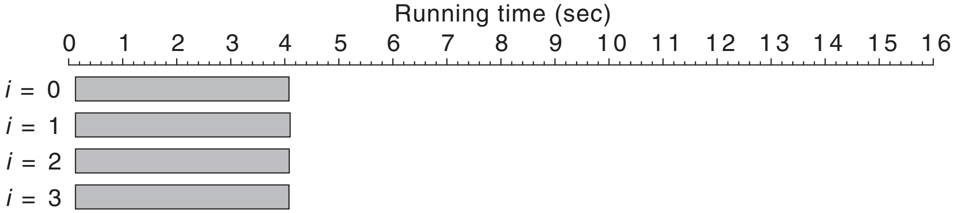


**Figure 4.3** Program1Smp execution timeline, SMP parallel computer

To be precise, the speedup (the reduction factor) was 15342/4098 = 3.744. The speedup was somewhat less than 4 because of overhead in the parallel version that is not present in the sequential version. With Program1Smp, the first subroutine call didn't begin until 125 msec after the program started. During this time, the program was occupied in creating the parallel team and parallel region objects, starting up the parallel team threads, and executing the parallel region's `run()` method—work that the sequential program didn't have to do.

This illustrates another central theme of parallel program design: *Parallelism is not free.* The benefit of speedup or sizeup comes with a price of extra overhead that is not needed in a sequential program. The name of the game is to minimize this extra overhead.

## 4.5 Running on a Regular Computer

Although intended to run on a parallel computer, Program1Seq and Program1Smp are perfectly happy to run on a nonparallel computer. In fact, one benefit of programming in Parallel Java is that you can develop and test parallel programs on any computer, and then you can shift to a parallel computer when the program is debugged and ready for usage.

Let's look at what happens when we run these programs on a regular computer. This was a nonparallel computer with a 1.6 GHz Intel Pentium CPU and 512 MB of main memory. The sequential Program1Seq program printed the following:

```
$ java Program1Seq 1000000000000037 1000000000000091 \
  1000000000000159 1000000000000187
i = 0 call start = 0 msec
i = 0 call finish = 1594 msec
i = 1 call start = 1594 msec
i = 1 call finish = 2881 msec
i = 2 call start = 2881 msec
i = 2 call finish = 4165 msec
i = 3 call start = 4165 msec
i = 3 call finish = 5450 msec
```

The timeline (Figure 4.4) shows the typical pattern of sequential execution.

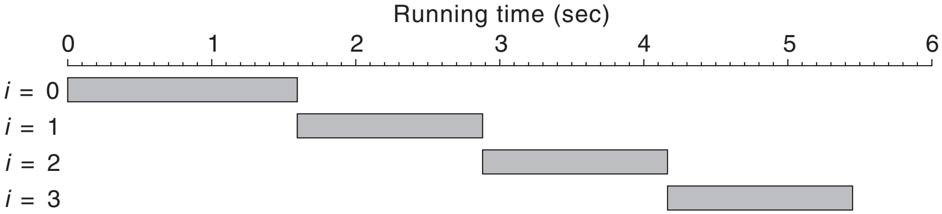


**Figure 4.4** Program1Seq execution timeline, regular computer

Suppose we run the multithreaded Program1Smp program on the regular computer. Because each subroutine call will now run in a different thread, we would expect all the subroutine calls to start at roughly the same time near the beginning of the program. But because all the threads will share the same processor, and the processor will execute one thread at a time and switch to another thread every so often, we would expect the overall running time to be about the same as the sequential program. Here is what the parallel program printed.

```
$ java Program1Smp 1000000000000037 1000000000000091 \
  1000000000000159 1000000000000187
i = 0 call start = 21 msec
i = 0 call finish = 5190 msec
```

```
i = 1 call start = 71 msec
i = 1 call finish = 5053 msec
i = 2 call start = 91 msec
i = 2 call finish = 5134 msec
i = 3 call start = 14 msec
i = 3 call finish = 4981 msec
```

The timeline for this run (Figure 4.5) is about what we expected—except for one thing. The overall running time was 260 msec shorter for the four-thread parallel version than for the single-thread sequential version. We got a slight but noticeable speedup when we went from one thread to four threads on the regular computer. How can this be, when there was only one processor?

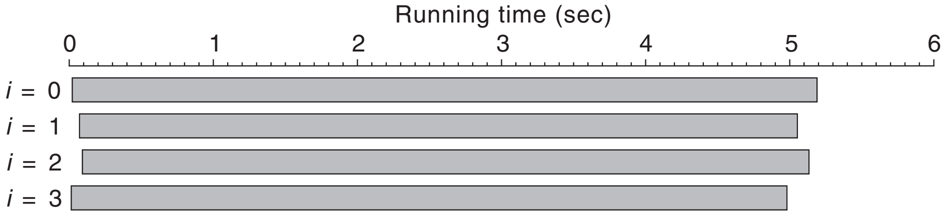


**Figure 4.5** Program1Smp execution timeline, regular computer

The reason has to do with how the JVM works. A modern JVM includes a **just-in-time (JIT) compiler** that converts the Java bytecode instructions into native machine code instructions as the program runs. The JVM then executes the machine code directly instead of interpreting the Java bytecode; this greatly increases the program's execution speed. Furthermore, a modern JVM monitors which sections of bytecode are executed most frequently and compiles just those sections to machine code, leaving the remaining sections as interpreted bytecode. This avoids spending the time it would take to compile infrequently used sections of bytecode. (Sun Microsystems refers to this as a **HotSpot JVM**.) However, it takes a certain amount of execution before the JVM detects that the `isPrime()` subroutine is a hot spot and compiles it to machine code. With four threads all calling the subroutine at the same time, the JVM can detect the hot spot, and compile it to machine code, sooner in the parallel version than in the sequential version. This allows more of the parallel version's running time to be executed in the faster machine code mode, thus reducing the parallel version's running time compared to the sequential version. (To verify that this is in fact what's going on, try running both versions with the JIT compiler disabled; the parallel version then invariably takes longer than the sequential version due to the parallel version's extra overhead.) We will see further instances of how the JVM's behavior influences program performance as we study parallel programming in Java.

## 4.6 The Rest of the Book

Let's step back and look at what we've done. We started with a problem statement. We wrote a sequential program and a parallel program to solve the problem. The parallel program illustrated both general parallel programming techniques (in this case, achieving repetition via multiple threads) and specific Parallel

Java features (parallel team and parallel region). Then we ran the sequential and parallel programs, measured their running times, and gained some insight about parallel programming by comparing the programs' performance.

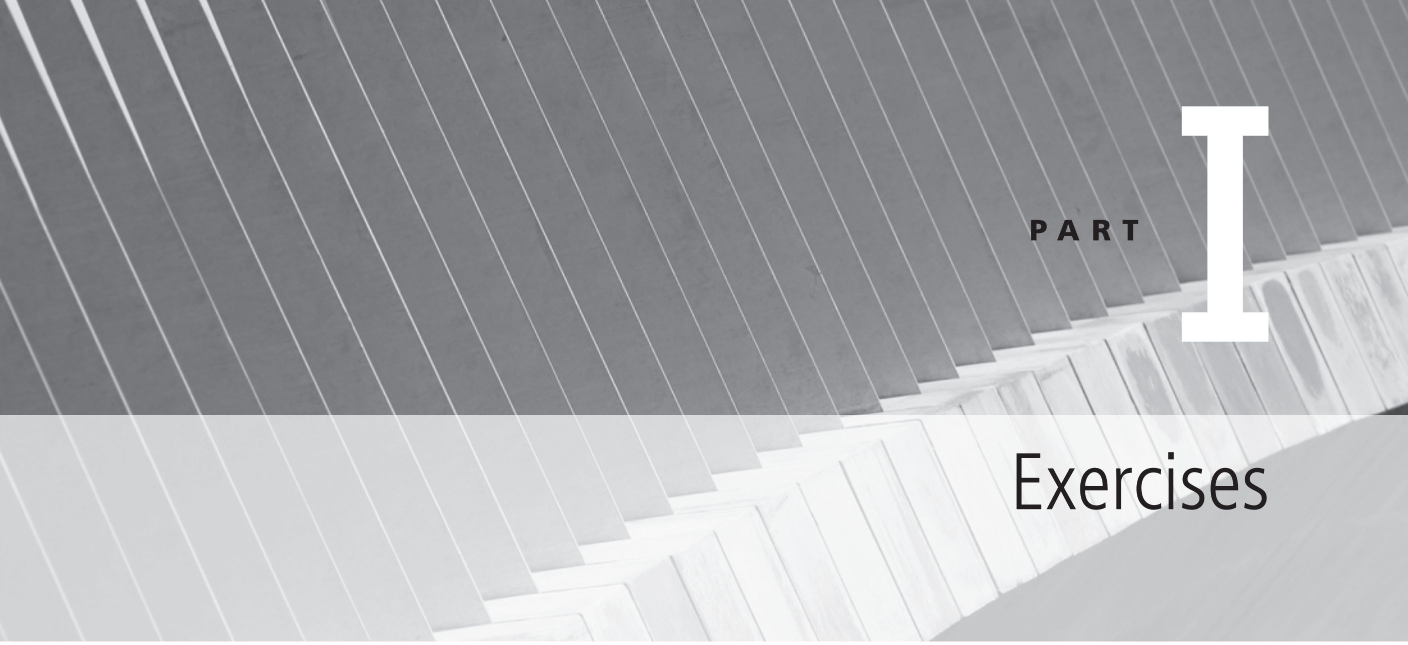
The rest of the book will be much the same—solving a series of problems that are chosen to illustrate various parallel programming techniques and studying the programs' performance measurements. In Part II, we will begin with SMP parallel programs, because those are quite similar to regular sequential programs. Then, in Part III, we will move on to cluster parallel programs, which are a bit more different from regular sequential programs due to the explicit message passing that is needed. In Part IV, we will combine techniques for SMP parallel programming and techniques for cluster parallel programming to write hybrid parallel programs. While the problems we solve in Parts II through IV will be interesting and perhaps fun, they were chosen solely for pedagogical reasons—to illustrate parallel programming techniques—and are not necessarily problems with any great significance in the real world. Finally, in Part V, we will apply the techniques we've learned to solve some *real-world* problems using parallel computing.

## 4.7 For Further Information

On the HotSpot JVM, and performance tuning of Java programs in general:

- Steve Wilson and Jeff Kesselman. *Java Platform Performance: Strategies and Tactics*. Addison-Wesley, 2000. Available online at: http://java.sun.com/docs/books/performance/
- Java Performance Documentation. http://java.sun.com/docs/performance/index.html

*This page intentionally left blank*



# PART I

# Exercises

*Exercises 1–2.* The year is 1944. World War II is at its height. The Computer Department at your company has 50 state-of-the-art computers all together in a big room. Each "computer" is a young lady with a Friden automatic electromechanical calculator that can do addition, subtraction, multiplication, and division. Your task is to prepare a table of the function:

$$f(x) = ax^2 + bx + c \tag{1}$$

Collection of the author

Friden automatic electromechanical calculator for given constant values $a$, $b$, and $c$ and for 1,020 values of $x$: $x = 0.1, 0.2, 0.3, \ldots, 101.8, 101.9, 102.0$. This table will be used for computing the trajectories of artillery shells on the battlefield.

1. Describe a parallel algorithm to solve the problem in the shortest possible time using the full resources of the Computer Department.
2. Which parallel design pattern or patterns does this problem exhibit?

*Exercises 3–4.* Here's another problem for the Computer Department. Each computer has a sheet of paper with a list of all the prime numbers less than or equal to 1000; there are 168 primes in the list. Your problem is to determine whether the number $N = 988027$ is prime. The algorithm is as follows: Divide $N$ by each prime in the list; if any prime in the list divides $N$ with no remainder, then $N$ is not prime; otherwise, $N$ is prime. The algorithm always goes through the entire list of primes. (Note: If $N$ is not prime, then $N$ must have a prime factor $P \le \sqrt{N} < 1000$, so $P$ must be in the list of primes and the algorithm is guaranteed to find $P$.)

3. Describe a parallel version of the preceding algorithm to solve the problem in the shortest possible time using the full resources of the Computer Department.
4. Which parallel design pattern or patterns does this problem exhibit?

*Exercises 5–6.* Here's another problem for the Computer Department. Multiply two matrices $A$ and $B$ to get the product matrix $C = A \cdot B$. Each matrix has 12 rows and 12 columns of numbers. The elements of the two matrices $A$ and $B$ have been printed on a sheet of paper, and each computer has a copy. The formula for computing the element at row $i$, column $j$ of the product is:

$$C_{ij} = \sum_{k=1}^{12} A_{ik} \cdot B_{kj} \tag{2}$$

5. Describe a parallel algorithm to solve the problem in the shortest possible time using the full resources of the Computer Department.
6. Which parallel design pattern or patterns does this problem exhibit?

*Exercises 7–8.* Here's another problem for the Computer Department. Compute the mean of 256 given numbers $x_1$ through $x_{256}$; that is:

$$\frac{1}{256} \sum_{i=1}^{256} x_i \tag{3}$$

7. Describe a parallel algorithm to solve the problem in the shortest possible time using the full resources of the Computer Department.
8. Which parallel design pattern or patterns does this problem exhibit?

*Exercises 9–10.* Here's another problem for the Computer Department. Your company's Sales Department has 360 salesmen (in 1944, they were all males) selling floor brushes, hairbrushes, toothbrushes, and scrub brushes door-to-door. Each salesman has submitted his quarterly sales report showing the number of floor brushes, hairbrushes, toothbrushes, and scrub brushes he sold. Your task is to calculate each salesman's total commission. The commission is 50 cents for each floor brush, 25 cents for each hairbrush, 10 cents for each toothbrush, and one dollar for each scrub brush.

9. Describe a parallel algorithm to solve the problem in the shortest possible time using the full resources of the Computer Department.
10. Which parallel design pattern or patterns does this problem exhibit?

*Exercises 11–16.* For each of the following parallel computing systems, do some research, describe how the hardware and software are designed, and state which parallel design pattern or patterns each system's parallel program exhibits.

11. Weather Research and Forecasting (WRF).
12. Community Climate System Model (CCSM).
13. Basic Local Alignment Search Tool (BLAST).
14. Great Internet Mersenne Prime Search (GIMPS).
15. SETI@home.
16. Deep Blue.

*Exercises 17–18.* A cluster parallel program needs to send a message consisting of 4,000 integers (type `int`). Each `int` occupies 4 bytes. Each byte is 8 bits.

17. How long will it take to send the message if the cluster backend network is an Ethernet with a bandwidth of 1 Gbps and a latency of 150 $\mu$sec?
18. How long will it take to send the message if the cluster backend network is an InfiniBand with a bandwidth of 48 Gbps and a latency of 2 $\mu$sec?

*Exercises 19–20.* A parallel program calculates an $n \times n$-element matrix, where each matrix element is a double-precision floating-point number (type `double`). Each `double` occupies 8 bytes.

19. How large a matrix can the parallel program calculate if run on an SMP parallel computer with 8 GB of main memory available to store the matrix? (1 GB = $2^{30}$ bytes.)
20. How large a matrix can the parallel program calculate if run on a 100-node cluster parallel computer with 256 MB of main memory on each node available to store the matrix? (1 MB = $2^{20}$ bytes.)

21. Run Program1Seq and Program1Smp with a different number of computations (command-line arguments), say 8 or 16. What are the programs' running times on an SMP computer? On a regular computer? What do you discover by examining the running times?

22. Run Program1Seq and Program1Smp so the computations take a different amount of time (that is, specify different values as command-line arguments). What are the programs' running times on an SMP computer? On a regular computer? What do you discover by examining the running times?

23. Suppose Program1Smp, when run on an SMP computer, did not experience the overhead between the start of the program and the start of the actual computations, but otherwise the computations took the same time as listed in Chapter 4, Section 4.4. What would the speedup be? How close would that be to the ideal?

24. Write sequential and SMP parallel versions of a program that does a given number of computations, where each computation consists of printing the message `"Hello world <i>"`, where `<i>` is the computation number starting from 0. The output should look like this:

    ```
    Hello world 0
    Hello world 1
    Hello world 2
    ...
    ```

    Run your programs on an SMP computer and on a regular computer with different numbers of computations. Do you notice anything strange happening? If so, why is it happening?

# PART II

# SMPs

*This page intentionally left blank*

# CHAPTER 5

# Massively Parallel Problems

in which we encounter a cryptographic problem requiring an enormous amount of computation; we build a sequential program to solve the problem; we reflect on how a parallel program could solve the problem; and we learn why such problems are called massively parallel

# CHAPTER 5 Massively Parallel Problems

## 5.1 Breaking the Cipher

To conceal passwords, credit card numbers, and other sensitive information from prying eyes while e-mail messages and Web pages traverse the public Internet, the information is **encrypted**. Nowadays encryption is done using a **block cipher**, such as the U.S. Government's **Advanced Encryption Standard (AES)** (Figure 5.1).

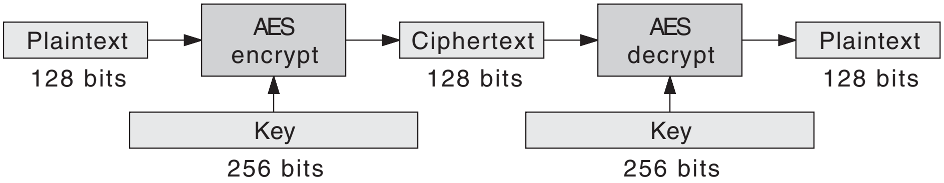


**Figure 5.1** Encryption and decryption using AES

Here's how one person named Alice uses a block cipher to send an encrypted message to another person named Bob. (In cryptographic lore, the two parties in a secure communication are always named Alice and Bob.) Alice feeds the original message, called the **plaintext**, into the AES **encryption function**. For AES, the plaintext is a block of 128 bits. The encryption function's output is another block of 128 bits, called the **ciphertext**. (AES is called a "block" cipher because it converts a *block* of plaintext into an equal-sized *block* of ciphertext.) The ciphertext is random-seeming gibberish that reveals nothing about the plaintext. Alice can safely send the ciphertext over a public network without fear that Eve, who is eavesdropping on the network traffic, will discover the sensitive information in the plaintext. Upon receiving the ciphertext, Bob feeds the ciphertext into the AES **decryption function**, which converts the ciphertext back into the original plaintext.

The security of block cipher encryption rests in the **key**, which is an input parameter to the encryption function and the decryption function. For AES, the key is a 256-bit value. The same plaintext, encrypted with a different key, will yield a different ciphertext. To recover the original plaintext from the ciphertext, the decryption function must use the same key as the encryption function. Because knowledge of the key would let Eve decrypt ciphertext messages, the key must be a **secret key** known only to Alice and Bob.

One way that Eve could breach the secure communication is to find the key Alice and Bob are using. One way that Eve could find the key is a **known plaintext attack**. Eve somehow manages to obtain both a plaintext block $p$ and the ciphertext block $c$ that is the result of encrypting $p$ with some secret key $k$. Eve then uses her knowledge of these corresponding $p$ and $c$ values to deduce the value of $k$. One way to find $k$ is an **exhaustive search**. Eve starts with $k = 0$, feeds $p$ and $k$ into the encryption function, and checks whether the ciphertext that comes out is equal to *c*. If so, Eve has found the correct value for *k*. Otherwise, Eve repeats the process with *k* = 1, *k* = 2, and so on until she is successful.

Block cipher key sizes are chosen to make exhaustive key searches impractical. To find an AES key this way, Eve has to perform on the order of $2^{256}$, or $10^{77}$, encryptions. Long before Eve has found the key, the universe will have come to an end.

However, suppose Eve knows *some* of the key. Perhaps Alice and Bob were careless and revealed the values of 232 bits of the 256-bit key. Then Eve only has to do $2^{24}$, or 16 million, encryptions to find the complete key. That's doable.

This, then, is our first problem: Write a program for an **AES partial key search**. The program's inputs are a plaintext block *p*, a ciphertext block *c*, and a portion of the key *k* that was used to produce *c* from *p*. The values of the known key bits are given, along with the number of missing bits. The program's output is the complete key. The program does an exhaustive search over all possible values for the missing key bits.

## 5.2 Preparing the Input

Let's do our AES partial key search on a realistic example. First, we need a random 256-bit key. However, it's a bad idea to use a **pseudorandom number generator (PRNG)** such as class java.util.Random to generate the key. The problem is that class java.util.Random has only 48 bits of internal state, from which it generates random values. If the secret key came from class java.util.Random, Eve would have to search only the $2^{48}$ possible internal state values to find the key, not the $2^{256}$ possible key values.

Rather than use a PRNG, we should use an **entropy source** to generate a random key. Most Unix and Linux kernels have a special device file, `/dev/random`, that provides an entropy source. The kernel accumulates "randomness," or entropy, into this file from the random times at which certain events occur, such as keystrokes, mouse movements, disk block accesses, and network packet receptions. Then, as a program reads this file, the kernel uses the accumulated entropy to return truly random bytes. In contrast, a PRNG generates only *pseudo*-random values using a deterministic formula.

Here is a program that prints a random 256-bit key in hexadecimal. To access the platform-dependent entropy source in a portable manner, we use the `getSeed()` method of class java.security.SecureRandom.

```
package edu.rit.smp.keysearch;
import edu.rit.util.Hex;
import java.security.SecureRandom;
public class MakeKey
   {
   public static void main
      (String[] args)
      throws Exception
      {
      System.out.println (Hex.toString (SecureRandom.getSeed (32)));
      }
   }
```

And here is an example of what the MakeKey program prints.

```
$ java edu.rit.smp.keysearch.MakeKey
26ab7c3cb314cb3eed163e1bb9a65aa0e7a2261fb7139e75412d4cf44719520b
```

Next, we need a plaintext-ciphertext pair to use for our known plaintext attack. Here is a program that creates just such a pair. The program takes three command-line arguments: a message string to encrypt, the encryption key (generated by the MakeKey program), and *n*, the number of key bits for which to search. The program prints the plaintext block (a 128-bit hexadecimal number), the ciphertext block (a 128-bit hexadecimal number), the partial key with the *n* least-significant bits set to 0 (a 256-bit hexadecimal number), and *n*, the number of key bits for which to search. To do the encryption, the program uses an instance of class AES256Cipher from the Parallel Java Library.

```java
package edu.rit.smp.keysearch;
import edu.rit.crypto.blockcipher.AES256Cipher;
import edu.rit.util.Hex;
public class Encrypt
   {
   public static void main
      (String[] args)
      throws Exception
      {
      // Parse command line arguments.
      if (args.length != 3) usage();
      String message = args[0];
      byte[] key = Hex.toByteArray (args[1]);
      int n = Integer.parseInt (args[2]);

      // Set up plaintext block.
      byte[] msg = message.getBytes();
      byte[] block = new byte [16];
      System.arraycopy
         (msg, 0, block, 0, Math.min (msg.length, 16));
      System.out.println (Hex.toString (block));

      // Encrypt plaintext.
      AES256Cipher cipher = new AES256Cipher (key);
      cipher.encrypt (block);
      System.out.println (Hex.toString (block));

      // Wipe out n least significant bits of the key.
      int off = 31;
      int len = n;
      while (len >= 8)
```

```
         {
         key[off] = (byte) 0;
         -- off;
         len -= 8;
         }
      key[off] &= mask[len];
      System.out.println (Hex.toString (key));
      System.out.println (n);
      }

   private static final int[] mask = new int[]
      {0xff, 0xfe, 0xfc, 0xf8, 0xf0, 0xe0, 0xc0, 0x80};
   }
```

And here is an example of what the Encrypt program prints. (The Java command stretches across two lines.) The Encrypt program's output will become the actual key-searching program's input.

```
$ java edu.rit.smp.keysearch.Encrypt "Hello, world!" \
  26ab7c3cb314cb3eed163e1bb9a65aa0e7a2261fb7139e75412d4cf44719520b 20
48656c6c6f2c20776f726c6421000000
af3afe16ce815ad209f34b009da37e58
26ab7c3cb314cb3eed163e1bb9a65aa0e7a2261fb7139e75412d4cf447100000
20
```

## 5.3 Sequential Key Search Program

Here is the FindKeySeq program, which takes the Encrypt program's outputs as command-line arguments—the plaintext, the ciphertext, the partial key, and *n*, the number of key bits for which to search. The FindKeySeq program performs an exhaustive search over all the missing key bits and prints the complete key. It is a sequential program (no parallelism); later, we will modify it to make a parallel program.

We follow the convention that variables used throughout the main program are declared as static fields of the main program class, rather than local variables of the `main()` method. The reason for this will become clear when we write the parallel program.

```
package edu.rit.smp.keysearch;
import edu.rit.crypto.blockcipher.AES256Cipher;
import edu.rit.util.Hex;
public class FindKeySeq
   {
   // Command line arguments.
   static byte[] plaintext;
   static byte[] ciphertext;
   static byte[] partialkey;
   static int n;

   // Variables for doing trial encryptions.
   static int keylsbs;
   static int maxcounter;
   static byte[] foundkey;
   static byte[] trialkey;
   static byte[] trialciphertext;
   static AES256Cipher cipher;

   /**
    * AES partial key search main program.
    */
   public static void main
      (String[] args)
      throws Exception
      {
      // Start timing.
      long t1 = System.currentTimeMillis();

      // Parse command line arguments.
      if (args.length != 4) usage();
      plaintext = Hex.toByteArray (args[0]);
      ciphertext = Hex.toByteArray (args[1]);
      partialkey = Hex.toByteArray (args[2]);
      n = Integer.parseInt (args[3]);

      // Make sure n is not too small or too large.
      if (n < 0)
         {
         System.err.println ("n = " + n + " is too small");
         System.exit (1);
         }
      if (n > 30)
         {
         System.err.println ("n = " + n + " is too large");
         System.exit (1);
         }
```

The variable `keylsbs` holds the least-significant 32 bits of the partial key from the command line; this value has *n* low-order zero bits.

```
// Set up variables for doing trial encryptions.
keylsbs =
   ((partialkey[28] & 0xFF) << 24) |
   ((partialkey[29] & 0xFF) << 16) |
   ((partialkey[30] & 0xFF) <<  8) |
   ((partialkey[31] & 0xFF)      );
```

To search over the missing key bits, we run an integer counter from 0 to $2^n - 1$. The variable `maxcounter` holds the upper bound for the counter. To be certain that this upper bound fits in a variable of type `int`, we ensure that $n$ lies in the range 0 through 30. The variable `trialkey` is for holding the encryption key we are currently trying; this starts out the same as the partial key from the command line. The variable `trialciphertext` is for holding the output ciphertext from the current encryption. The cipher object performs the actual encryptions.

```
maxcounter = (1 << n) - 1;
trialkey = new byte [32];
System.arraycopy (partialkey, 0, trialkey, 0, 32);
trialciphertext = new byte [16];
cipher = new AES256Cipher (trialkey);
```

Now we can run the search loop, with the counter going from 0 to $2^n - 1$. To set up the trial encryption key, we combine the counter value with the least significant bits of the partial key (`keylsbs`) using bitwise-or (the | operator). The counter fills in the missing low-order bits of the partial key, thus creating the complete trial key. We use the cipher object to encrypt the plaintext with this trial key.

```
// Try every possible combination of low-order key bits.
for (int counter = 0; counter < maxcounter; ++ counter)
   {
   // Fill in low-order key bits.
   int lsbs = keylsbs | counter;
   trialkey[28] = (byte) (lsbs >>> 24);
   trialkey[29] = (byte) (lsbs >>> 16);
   trialkey[30] = (byte) (lsbs >>>  8);
   trialkey[31] = (byte) (lsbs       );

   // Try the key.
   cipher.setKey (trialkey);
   cipher.encrypt (plaintext, trialciphertext);
```

If the resulting ciphertext equals the input ciphertext, we have found the correct key, and we store a copy of it in the variable `foundkey`. At this point, we could exit the loop, because further trials are pointless. However, for now, we will stay in the loop; the program will always try all $2^n$ key values. (We will need this behavior later, when we study the parallel version's running time.)

```
         // If the result equals the ciphertext, we found the key.
         if (match (ciphertext, trialciphertext))
            {
            foundkey = new byte [32];
            System.arraycopy (trialkey, 0, foundkey, 0, 32);
            }
         }

      // Stop timing.
      long t2 = System.currentTimeMillis();

      // Print the key we found.
      System.out.println (Hex.toString (foundkey));
      System.out.println ((t2-t1) + " msec");
      }

   /**
    * Returns true if the two byte arrays match.
    */
   private static boolean match
      (byte[] a,
       byte[] b)
      {
      boolean matchsofar = true;
      int n = a.length;
      for (int i = 0; i < n; ++ i)
         {
         matchsofar = matchsofar && a[i] == b[i];
         }
      return matchsofar;
      }
   }
```

Here is an example of the FindKeySeq program's output. (The Java command stretches across five lines.) The inputs (command-line arguments) are what the Encrypt program generated—the plaintext block, the ciphertext block, the partial key with the low-order bits missing, and the number of missing key bits. The FindKeySeq program prints the complete key that it found, as well as the running time.

```
$ java edu.rit.smp.keysearch.FindKeySeq \
  48656c6c6f2c20776f726c6421000000 \
  af3afe16ce815ad209f34b009da37e58 \
  26ab7c3cb314cb3eed163e1bb9a65aa0e7a2261fb7139e75412d4cf447100000 \
  20
26ab7c3cb314cb3eed163e1bb9a65aa0e7a2261fb7139e75412d4cf44719520b
2936 msec
```

## 5.4 Transitioning to a Parallel Program

The AES partial key search problem is an agenda parallel problem, where the agenda items are to try all possible values of the missing key bits: “Try $k = 0$,” “Try $k = 1$,” . . . , “Try $k = 2^n - 1$.” We are not interested in all the tasks’ results, but only in the key for the one task that succeeded. In this problem, the results of one task do not in any way affect the results of the other tasks; there are no sequential dependencies between tasks. Putting it an other way, none of the tasks produces any result needed by any other task.

The FindKeySeq program performs the agenda items by doing a loop over all the missing key bits from 0 to $2^n - 1$. In the sequential program, the loop iterations are performed one at a time, in order. However, because there are no sequential dependencies between tasks, the loop iterations do not have to be performed in sequence—as we saw with the introductory program in Chapter 4. The loop iterations can be performed all at once, in parallel. In fact, if we had a parallel computer with $2^n$ processors, we could find the answer in the same amount of time as the sequential program would take to try just one key. For this reason, a problem such as AES partial key search—one where we can do all the computations *en masse,* with no dependencies between the computations—is called a **massively parallel problem**. It is also sometimes called an **embarrassingly parallel problem**; there’s so much parallelism, it’s embarrassing!

In Chapter 4, we saw how to execute a parallel program with each computation in a separate thread and each thread running on its own processor. If $n$ is a small number, such as 2 or 4, we’d have no trouble finding a parallel computer with $2^n$ processors. But this approach breaks down if $n$ is a more interesting number such as 24 or 28. What we need is an approach somewhere in the middle of the two extremes of doing all the computations in sequence in a single thread and doing each computation in parallel in its own thread.

Suppose we are solving the AES partial key search program on a parallel computer with $K$ processors. Then we will set up a parallel program with $K$ threads. Like the sequential program, each thread will execute a loop to do its computations. However, each thread’s loop will go through only a *subset* of the total set of computations—$2^n/K$ of them, to be exact. The set of computations will be *partitioned* among the $K$ threads; multiple tasks will be *clumped* together and executed by one of the $K$ threads.

As a specific example, suppose $n$ is 20; then there are 1,048,576 keys to search. Suppose $K$ is 10; then six of the threads will do 104,858 iterations each and the other four threads will do 104,857 iterations each. (The per-thread iteration counts differ because the total number of iterations is not evenly divisible by $K$.) The range of counter values in each thread will be as follows:

| *Thread* | *Lower Bound* | *Upper Bound* | *Thread* | *Lower Bound* | *Upper Bound* |
|---:|---:|---:|---:|---:|---:|
| 0 | 0 | 104857 | 5 | 524290 | 629147 |
| 1 | 104858 | 209715 | 6 | 629148 | 734004 |
| 2 | 209716 | 314573 | 7 | 734005 | 838861 |
| 3 | 314574 | 419431 | 8 | 838862 | 943718 |
| 4 | 419432 | 524289 | 9 | 943719 | 1048575 |

The Parallel Java Library has classes that let you program **parallel loops**, where the total set of loop iterations is divided among a group of threads in the preceding manner. Chapter 6 will introduce these features of Parallel Java. In Chapter 7, we will use Parallel Java to convert the FindKeySmp program to a parallel program.

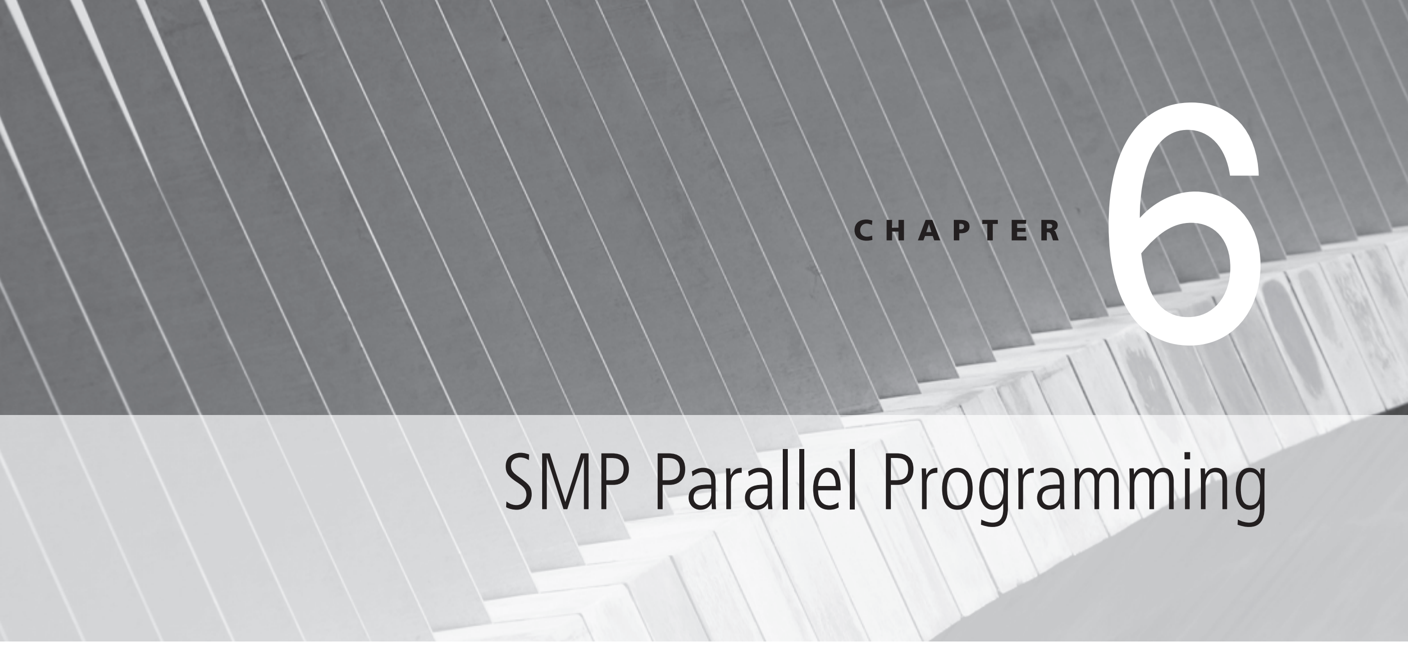
## 5.5 For Further Information

On cryptography, block ciphers, and ways of attacking them:

- Douglas R. Stinson. *Cryptography: Theory and Practice, 3rd Edition*. Chapman & Hall, 2005.
- Niels Ferguson and Bruce Schneier. *Practical Cryptography*. Wiley Publishing, 2003.
- Charlie Kaufman, Radia Perlman, and Mike Speciner. *Network Security: Private Communication in a Public World, 2nd Edition*. Prentice Hall PTR, 2002.
- Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1997.
- Bruce Schneier. *Applied Cryptography, Second Edition*. John Wiley & Sons, 1996.

On the Advanced Encryption Standard in particular:

- *Advanced Encryption Standard (AES)*. Federal Information Processing Standards Publication 197. November 26, 2001. http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf



# CHAPTER 6

# SMP Parallel Programming

in which we take a closer look at the principal constructs for SMP parallel programming; we learn how to declare shared and non-shared variables; and we gain some insight into how a Parallel Java program achieves parallelism on an SMP parallel computer

# CHAPTER 6 SMP Parallel Programming

## 6.1 Parallel Team

A sequential program to do some computation, as embodied in the main program class's `main()` method, looks like this:

Initial setup

Execute the computation

Clean up

To change this to an SMP parallel program, we change the middle step to use a **parallel team**:

Initial setup

Create a parallel team

Have the parallel team execute the computation

Clean up

In Parallel Java, you get a parallel team by creating an instance of class ParallelTeam. The parallel team comprises a certain number of **threads**, which will be the ones to carry out the computation in parallel. You have three choices for specifying the number of threads in the team. You can specify the number of threads at compile time as the constructor argument.

```
    new ParallelTeam(4); // Create a team of four threads
```

You can specify the number of threads at run time by using the no-argument constructor.

```
    new ParallelTeam();
```

In this case, you specify the number of threads by defining the `"pj.nt"` Java system property when you run the program. You can do this by including the `"-Dpj.nt"` flag on the Java command line. For example,

```
$ java -Dpj.nt=4 . . .
```

specifies four threads. Finally, if you use the no-argument constructor and you do not define the `"pj.nt"` property, Parallel Java chooses the number of parallel team threads automatically to be the same as the number of processors on the computer where the program is running. (Parallel Java discovers the number of processors by calling the `Runtime.availableProcessors()` method, which is part of the standard Java platform.)

Normally, you will let Parallel Java choose the number of threads automatically at run time, so as to get the maximum possible degree of parallelism. Sometimes you will specify the number of threads explicitly at run time—for example, when you are measuring the parallel program's performance as a function of the number of processors. Certain parallel programming patterns, such as "overlapping" (which we will study in Chapter 18), always use the same number of threads, and when utilizing these patterns, you specify the number of threads at compile time.

Once a parallel team has been created, *K* threads are in existence inside the parallel team object (Figure 6.1). These are in addition to the Java program's main thread, the thread that is executing the main program class's `main()` method and that created the parallel team object. Each parallel team thread has a **thread index** from 0 through *K*–1. Each thread is poised to execute the program's computation. However, these threads are hidden, and you do not manipulate them directly in your program. To get the threads to execute the program's computation, you use another Parallel Java class, namely class ParallelRegion.

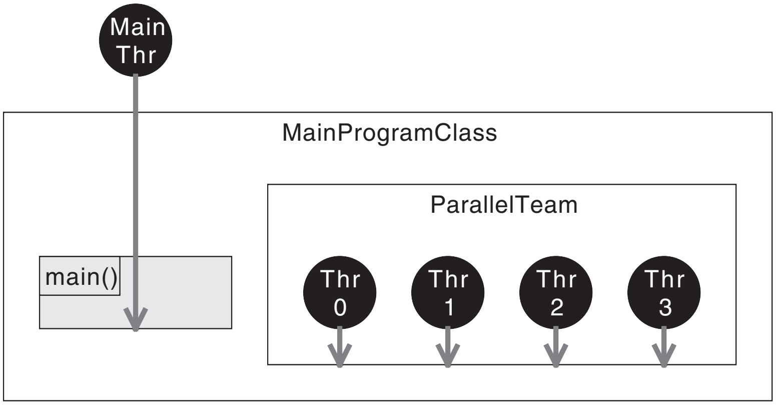


**Figure 6.1** A parallel team witk *K* = 4 threads

## 6.2 Parallel Region

Whereas a parallel team contains the *threads* that will carry out the program's computation, a **parallel region** contains the actual *code* for the computation. Class ParallelRegion is an abstract base class with three principal methods—`start()`, `run()`, and `finish()`—that you can override. To write the parallel portion of your program, define a subclass of class ParallelRegion and put the code for the parallel computation in the subclass's `start()`, `run()`, and `finish()` methods. To make the parallel team execute the code in the parallel region, create an instance of your ParallelRegion subclass and pass that instance to the parallel team's `execute()` method. A convenient coding idiom is to use an **anonymous inner class**.

```
new ParallelTeam().execute (new ParallelRegion()
  {
  public void start()
     {
     // Initialization code
     }
  public void run()
     {
     // Parallel computation code
     }
  public void finish()
     {
     // Finalization code
     }
  });
```

This code creates a new parallel team object, defines the ParallelRegion subclass as an anonymous inner class, creates an instance of the parallel region subclass, and passes the instance to the parallel team's `execute()` method.

Here's what happens inside the parallel team (Figure 6.2). Each parallel team thread is initially blocked and will not commence execution until signaled. The main thread, executing the program's `main()` method, calls the parallel team's `execute()` method. Inside the `execute()` method, the main thread calls the parallel region's `start()` method. When the `start()` method returns, the main thread signals each team thread to commence execution. The main thread then blocks.

All the team threads now proceed to call the parallel region's `run()` method. Here is where the parallel execution of the parallel region code happens. When the `run()` method returns, each team thread signals the main thread. At this point, the team threads block again, waiting for the team to execute another parallel region.

Once the main thread has been signaled by each of the team threads, the main thread resumes execution and calls the parallel region's `finish()` method. When the `finish()` method returns, the parallel team's `execute()` method also returns, and the main thread continues executing whatever comes afterward in the program's `main()` method.

It's important to emphasize that you don't have to do the preceding steps yourself. Parallel Java does it all for you. You merely have to write the code you want executed in the parallel region's `start()`, `run()`, and `finish()` methods. However, to design your parallel program properly, you have to understand what's going on "under the hood."

Focusing on the parallel region's methods in Figure 6.2, the sequence of execution is the following:

- The `start()` method is executed by a single thread.
- The `run()` method is executed by *K* threads simultaneously.
- The `finish()` method is executed by a single thread.

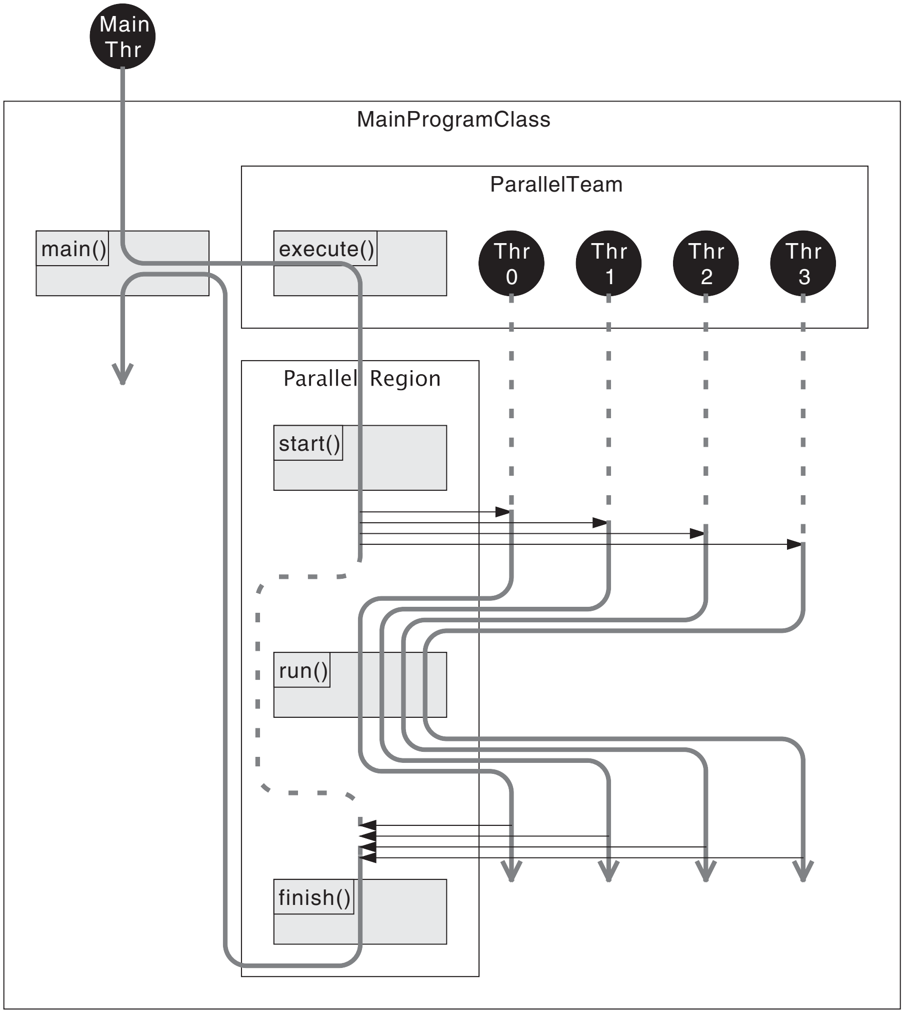


**Figure 6.2** A parallel team executing a parallel region

Thus, in the `start()` method, put any initialization code that must be executed in a single thread before the parallel computation starts. In the `run()` method, put the parallel computation code itself. In the `finish()` method, put any finalization code that must be executed in a single thread after the parallel computation finishes. If no initialization or finalization code is necessary, simply omit the `start()` or `finish()` method or both.

As already mentioned, when running a program with a parallel team and a parallel region on an SMP parallel computer, the JVM and the operating system are responsible for scheduling each thread to execute on a different processor. To see a parallel speedup, the parallel region's `run()` method must divide the computation among the *K* team threads—that is, among the *K* processors. With all processors executing simultaneously and each processor doing 1/*K* of the total work, the program should experience a speedup.

## 6.3 Parallel For Loop

Often, a program's computation consists of some number *N* of loop iterations. To divide the *N* loop iterations among the *K* threads (processors), you use yet another Parallel Java class, IntegerForLoop, which provides a **parallel for loop**. Class IntegerForLoop is an abstract base class with three principal methods—`start()`, `run()`, and `finish()`—that you can override. To write the parallel for loop, define a subclass of class IntegerForLoop and put the loop code in the subclass's `start()`, `run()`, and `finish()` methods. Then, in the parallel region's `run()` method, call the parallel region's `execute()` method, passing in the first loop index (inclusive), the last loop index (inclusive), and the IntegerForLoop subclass instance. Again, a convenient coding idiom is to use an anonymous inner class. Here is a parallel for loop with the index going from 0 to 99, inclusive.

```
new ParallelTeam().execute (new ParallelRegion()
  {
  public void run()
     {
     execute (0, 99, new IntegerForLoop()
        {
        public void start()
           {
           // Per-thread pre-loop initialization code
           }
        public void run (int first, int last)
           {
           // Loop code
           }
        public void finish()
           {
           // Per-thread post-loop finalization code
           }
        });
     }
  });
```

Here's what happens inside the parallel team (Figure 6.3). The parallel team threads are executing the parallel region's `run()` method simultaneously. Each team thread executes the following statement:

```
execute (0, 99, new IntegerForLoop()...);
```

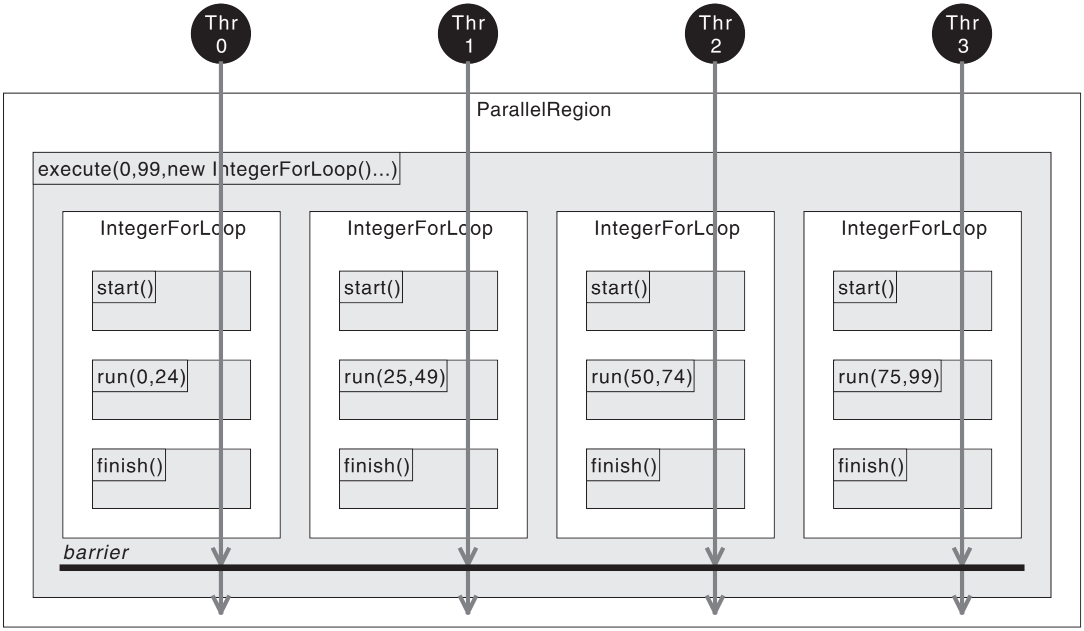


**Figure 6.3** A parallel team executing a parallel for loop

Each thread, therefore, first creates its own new instance of the IntegerForLoop subclass. Each thread then calls the parallel region's `execute()` method, passing in the loop index lower bound (0), the loop index upper bound (99), and the thread's own IntegerForLoop object. (All threads must pass in the *same* loop index lower and upper bounds, and these must be the bounds for the *whole* loop.) The parallel region's `execute()` method partitions the complete index range, 0–99 in this example, into *K* equal-sized **subranges**, or **chunks**, namely 0–24, 25–49, 50–74, and 75–99. Each thread now proceeds to call its own IntegerForLoop object's `start()`, `run()`, and `finish()` methods in sequence. However, each thread passes a different chunk to the `run()` method. Thread 0 passes the arguments first = 0 and last = 24, thread 1 passes the arguments 25 and 49, thread 2 passes the arguments 50 and 74, and thread 3 passes the arguments 75 and 99. After completing this sequence of execution through its IntegerForLoop object, each thread waits at a **barrier**. A barrier is a thread synchronization mechanism that ensures that none of the threads will proceed past the barrier until all the threads have arrived at the barrier. When the last thread finishes its portion of the parallel for loop and arrives at the barrier, like the final horse arriving at the starting gate for the Kentucky Derby, the barrier opens. Each thread resumes execution, returns from the parallel region's `execute()` method, and continues executing the code that comes after the parallel for loop in the parallel region's `run()` method.

As with the parallel region, it's important to emphasize that you don't have to do the preceding steps. Parallel Java does it all for you. You merely have to write the code you want executed in the parallel for loop's `start()`, `run()`, and `finish()` methods.

The purpose of the parallel for loop's `start()` and `finish()` methods is to do any necessary initialization within the parallel for loop before beginning the actual loop iterations, and to do any necessary finalization after finishing the loop iterations. If no initialization or finalization code is necessary, simply omit the `start()` or `finish()` method or both.

The parallel for loop's `run()` method's job is to execute the loop iterations for the chunk whose first and last loop indexes, inclusive, are passed in as arguments. Thus, the `run()` method typically looks like this.

```
public void run (int first, int last)
  {
  for (int i = first; i <= last; ++ i)
     {
     // Code for loop iteration i
     }
  }
```

Note that your code does not decide which chunk a particular call of the `run()` method will perform. The parallel region decides that. The `run()` method must do exactly the loop iterations specified by the `first` and `last` arguments, no more, no less.

Figure 6.3 shows where the parallel for loop's speedup comes from. Instead of executing 100 loop iterations in sequence as a regular program would do, the parallel program executes four chunks of 25 iterations in parallel, each chunk being executed by a different thread. Each thread (processor) does 1/*K* of the total work, resulting in a speedup.

Parallel Java also has classes for doing a parallel loop with an index of type `long`, and for doing a parallel loop over a collection of objects instead of a range of indexes. For further information, refer to the Parallel Java documentation.

## 6.4 Variables

Having looked at where to put the *code* for a Parallel Java program, let us turn our attention to where to put the *variable declarations* for a Parallel Java program.

Following the aforementioned coding idioms gives rise to a nested class structure (Figure 6.4). The parallel for loop class is nested inside the parallel region class, which in turn is nested inside the main program class.

```
public class MainProgramClass
  {
  // Shared variable declarations
  static int a;

  public static void main
     (String[] args)
     throws Exception
     {
     // Main program local variable declarations
     int b;
```

```
   new ParallelTeam().execute (new ParallelRegion()
      {
      public void run()
          {
          execute (0, 99, new IntegerForLoop()
             {
             // Per-thread variable declarations
             int c;

             public void run (int first, int last)
                {
                // Loop local variable declarations
                int d;
                for (int i = first; i <= last; ++ i)
                   {
                   // Code for loop iteration i
                   }
                }
             });
          }
      });
   }
}
```

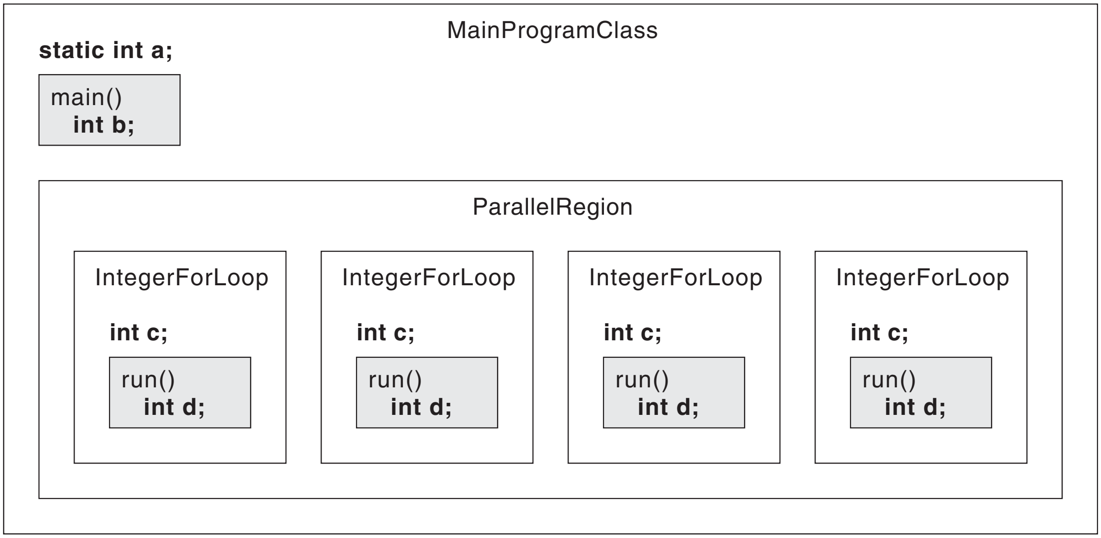


**Figure 6.4** Variable declarations in a Parallel Java program

Variables can be declared either as fields of these classes, or as local variables of the classes' methods. In practice, there are four categories of variables in a Parallel Java program; they differ in the places they are declared and in the ways they are accessed.

**Shared variables** (such as `a`) are declared as static fields of the main program class. Because they are *static* fields, there is only one instance of each shared variable. Furthermore, each shared variable can be accessed by code in the static `main()` method, by code in other static methods of the main program class, and by code in the parallel region and parallel for loop subclasses. Thus, the main program thread and all the parallel team threads can access every shared variable. These variables are called "shared" variables to emphasize that all the threads access the *same* instance of each variable. If one thread changes the value of a shared variable, all the other threads will see that new value.

**Per-thread variables** (such as `c`) are declared as instance fields of the parallel for loop subclass. Such variables can be accessed by code anywhere in the parallel for loop subclass. Because each thread creates its own instance of the parallel for loop subclass, each thread gets its own separate instances of the per-thread variables. If one thread changes the value of a per-thread variable, this will not affect the value of the corresponding per-thread variable in any other thread; per-thread variables are *not* shared. The parallel for loop's `start()` and `finish()` methods can be used to initialize and finalize these per-thread variables.

**Loop local variables** (such as `d`) are declared as local variables of the parallel for loop subclass's `run()` method. The loop control variable (such as `i`) is also a loop local variable. Loop local variables can be accessed only by code in the parallel for loop's `run()` method, and each thread gets its own separate instance of each loop local variable.

**Main program local variables** (such as `b`) are declared as local variables of the static `main()` method. Such variables are used only by the main thread executing the `main()` method. (Code inside the parallel region or parallel for loop cannot access main program local variables.)

An SMP parallel program is organized around a data structure or data structures located in **shared memory**, that is, in memory accessed by all the processors (threads) in the SMP parallel computer (Figure 6.5). In a result parallel program, the shared data structure may contain all the program's results. In an agenda parallel program, the shared data structure may contain the program's agenda items and their results. In a specialist parallel program, the shared data structure may contain certain tasks' outputs, which become other tasks' inputs. To get a parallel speedup, *all* the threads must access the *same* data structure, each thread working with a different piece of the data structure simultaneously. In a Parallel Java program, a data structure is made to reside in shared memory by declaring the data structure as a shared variable. The JVM then ensures that all the threads access the same instance of the variable.

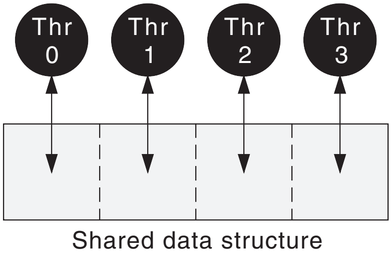


**Figure 6.5** Shared data structure, each thread accessing its own portion

If a variable is shared by multiple threads, however, we must make sure that the threads do not **conflict** with each other when they access the shared variable. A thread can do certain operations on a variable:

- A thread can **read** a variable; that is, the thread can examine the variable's state without changing its state.
- A thread can **write** a variable; that is, the thread can change the variable's state.
- A thread can **update** a variable; that is, the thread can read the variable's state, compute a new state based on the old state, and write the new state back into the variable.

A conflict can arise when two or more threads do certain operations on a variable at the same time:

- A **read-write conflict** can arise if one thread reads a variable at the same time as another thread writes or updates the variable; the reading thread may read an inconsistent state where part of the state is the old state and part of the state is the new state.
- A **write-write conflict** can arise if two threads write or update a variable at the same time; one thread's writes may wipe out some of the other thread's writes, again leading to an inconsistent state.
- However, there is no conflict if two threads read a variable at the same time.

If multiple threads cannot conflict when they access a shared variable, then we don't have to do anything special. But if multiple threads can conflict when they access a shared variable, we must **synchronize** the threads to eliminate the potential conflict. Parallel Java has several constructs for thread synchronization that we will study later. As we will see, thread synchronization adds overhead to the parallel program and can significantly increase the program's running time. Because the goal of parallel programming is to *reduce* the running time, we must carefully analyze each shared variable for thread conflicts and introduce thread synchronization only where it is absolutely needed.

A key aspect of designing a Parallel Java program, then, is to decide where in the program to declare each variable, depending on whether the variable does or does not need to be shared. A second key aspect is to decide whether and how to synchronize the multiple threads accessing each shared variable. This will be a recurring theme as we study SMP parallel programming in the chapters ahead.

With this introduction to Parallel Java's constructs for SMP parallel programming, we are ready to convert the sequential AES key search program from Chapter 5 to an SMP parallel program in Chapter 7.

## 6.5 For Further Information

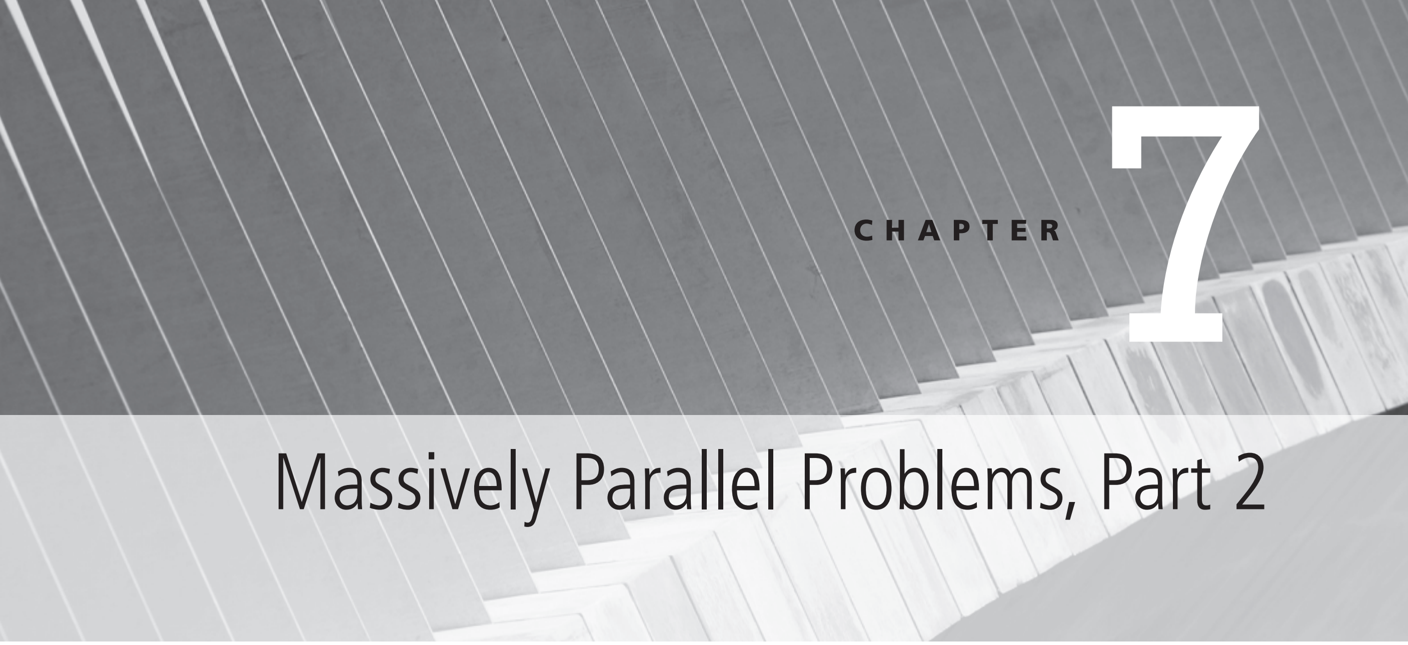
On the concepts of multithreading and thread synchronization, see any operating systems textbook, such as:

- A. Silberschatz, P. Galvin, and G. Gagne. *Operating System Concepts, 8th Edition*. John Wiley & Sons, 2009.

- A. Tanenbaum. *Modern Operating Systems, 3rd Edition*. Prentice-Hall, 2007.
- W. Stallings. *Operating Systems: Internals and Design Principles, 5th Edition*. Prentice-Hall, 2004.
- G. Nutt. *Operating Systems*, 3rd Edition. Addison-Wesley, 2003.

On the concepts of multithreaded programming and thread synchronization in Java:

- B. Goetz, T. Peierls, J. Bloch, J. Bowbeer, D. Holmes, and D. Lea. *Java Concurrency in Practice*. Addison-Wesley, 2006.
- D. Lea. *Concurrent Programming in Java: Design Principles and Patterns, 3rd Edition*. Addison-Wesley, 2006.
- Java Threads Tutorial.
  http://java.sun.com/docs/books/tutorial/essential/threads/ index.html



# CHAPTER 7

# Massively Parallel Problems, Part 2

in which we convert the sequential program for a massively parallel cryptographic problem into an SMP parallel program; and we learn how to apply Parallel Java’s SMP parallel programming features

# CHAPTER 7 Massively Parallel Problems, Part 2

## 7.1 AES Key Search Parallel Program Design

Now that we've been introduced to the constructs for writing SMP parallel programs in Parallel Java, let's return to the AES partial key search problem from Chapter 5. In that chapter, we built a sequential program that found the complete encryption key, given a plaintext, the corresponding ciphertext, and the partial key. Now we'll build an SMP parallel program that does the same thing.

The first step in designing the parallel version is to identify the computational code that will be executed in parallel. That's easy; it is the for loop in the middle of the main program. As we have already remarked, because this is a massively parallel problem, each computation (loop iteration) is independent of every other computation, and the iterations can all be done in parallel. Thus, the for loop will become a parallel for loop inside a parallel region.

The next step is to analyze the program's variables, decide which ones will and will not be shared by the parallel team threads, and decide where to declare each variable.

- `plaintext`, `ciphertext`, `partialkey`, `n`—A program's command-line arguments are typically used throughout the program. They will always be shared variables.
- `keylsbs`—This holds the least significant 32 bits of the partial key. It is written by the main program during the setup phase and read (but not written) by each thread during the parallel for loop. Because it is accessed both by the main program and by the threads, it will be a shared variable.
- `maxcounter`—This is the upper bound for the loop counter that ranges over all possible values for the missing key bits. It is written by the main program and read (but not written) during the parallel for loop. It, too, will be a shared variable.
- `foundkey`—This holds a copy of the key that was found to encrypt the plaintext correctly. It is written by one of the threads during the parallel for loop and read by the main program during the cleanup phase when the results are printed. It will be a shared variable.
- `trialkey`—This holds the complete key that the current loop iteration is using for its trial encryption. While the most significant bits of the trial key remain constant, the least significant bits are different on every loop iteration. Therefore, `trialkey` must be a per-thread variable. Each thread must have its own copy so that one thread will not overwrite another thread's trial key.

- `trialciphertext`—This receives the result of the current loop iteration's trial encryption. Again, this will be different on every loop iteration. It, too, will be a per-thread variable, so that one thread will not overwrite another thread's trial ciphertext.
- `cipher`—This is the AES block cipher object that does the actual encryption. It uses a different key on each loop iteration. It, too, will be a per-thread variable, so that one thread will not change the key of another thread's cipher object.
- `t1`, `t2`—These are used solely within the main program for timing measurements. They will remain local variables of the main program.

Now we need to take a second look at each of the shared variables and decide whether the threads can conflict with each other when accessing the shared variables. If a conflict is possible, we have to decide how to synchronize the threads. We don't have to worry about the per-thread variables, because only one thread will ever access those.

- `plaintext`, `ciphertext`, `partialkey`, `n`—These are **write once, read many (WORM)** variables. Once initialized from the command-line arguments, they are never written again, only read. Because multiple threads reading a shared variable do not conflict with each other, no synchronization is needed for these variables.
- `keylsbs`—WORM variable. No synchronization is needed.
- `maxcounter`—WORM variable. No synchronization is needed.
- `foundkey`—Because there are $2^{256}$ possible AES keys, but only $2^{128}$ possible ciphertext blocks, it must be the case that different keys yield the same ciphertext block for a given plaintext block. Therefore, it is possible for different threads to find a correct key and store it in the `foundkey` variable. Consequently, write-write conflicts are possible, and thread synchronization is needed. After the parallel region has finished, when the main program prints the contents of `foundkey`, only the main thread is executing; so no conflicts are possible and no synchronization is needed at this point.
- The remaining variables are per-thread variables or main program local variables and need no synchronization.

We have reached the conclusion that *this* program does not need to synchronize the threads when accessing the shared variables, except when writing the `foundkey` variable. It's important to emphasize that we are not merely omitting the synchronization out of laziness. We have carefully analyzed the program, decided where synchronization is and is not needed, and deliberately omitted any unnecessary synchronization, thereby eliminating the overhead that goes with it. When designing a parallel program, you must *always* analyze how your variables are accessed.

To prevent write-write conflicts over the `foundkey` variable, each thread will make a copy of the correct key in a temporary byte array, and then will store a reference to this byte array in the `foundkey` variable. Reads and writes of an array reference variable in Java are guaranteed to be **atomic**. If multiple threads try to read or write the variable simultaneously, the JVM ensures that each read or write operation finishes before the next read or write operation begins. Thus, the JVM itself synchronizes multiple threads writing the `foundkey` variable. It's important to emphasize that this synchronization happens only when writing the array *reference,* not the array *elements.* That's why each thread makes its own copy of the array elements first, and afterward writes the array reference into the `foundkey` variable.

Reads and writes of the following Java primitive types are guaranteed to be atomic: `boolean`, `byte`, `char`, `short`, `int`, and `float`. Reads and writes of object and array references are also guaranteed to be atomic. Reads and writes of the types `long` and `double` are not guaranteed to be atomic. In addition, *updates* of any type of variable—where the old value is read and a new value is computed and written back—are not guaranteed to be atomic. SMP parallel programs where multiple threads use `long` or `double` shared variables, or where multiple threads update shared variables, must synchronize the threads when they access such variables.

## 7.2 AES Key Search Parallel Program Code

Taking the foregoing design considerations into account, here is the code for the SMP parallel version of the AES key search program, class FindKeySmp.

```
package edu.rit.smp.keysearch;
import edu.rit.crypto.blockcipher.AES256Cipher;
import edu.rit.pj.IntegerForLoop;
import edu.rit.pj.ParallelRegion;
import edu.rit.pj.ParallelTeam;
import edu.rit.util.Hex;
public class FindKeySmp
   {
```

As in the sequential version, we declare the shared variables as static fields of the main program class.

```
   // Command line arguments.
   static byte[] plaintext;
   static byte[] ciphertext;
   static byte[] partialkey;
   static int n;

   // The least significant 32 bits of the partial key.
   static int keylsbs;

   // The maximum value for the missing key bits counter.
   static int maxcounter;
   // The complete key.
   static byte[] foundkey;

   /**
    * AES partial key search main program.
    */
   public static void main
      (String[] args)
      throws Exception
      {
      // Start timing.
      long t1 = System.currentTimeMillis();
```

In the main program's setup phase, we initialize the shared variables.

```
      // Parse command line arguments.
      if (args.length != 4) usage();
      plaintext = Hex.toByteArray (args[0]);
      ciphertext = Hex.toByteArray (args[1]);
      partialkey = Hex.toByteArray (args[2]);
      n = Integer.parseInt (args[3]);

      // Make sure n is not too small or too large.
      if (n < 0)
         {
         System.err.println ("n = " + n + " is too small");
         System.exit (1);
         }
      if (n > 30)
         {
         System.err.println ("n = " + n + " is too large");
         System.exit (1);
         }

      // Set up shared variables for doing trial encryptions.
      keylsbs =
         ((partialkey[28] & 0xFF) << 24) |
         ((partialkey[29] & 0xFF) << 16) |
         ((partialkey[30] & 0xFF) <<  8) |
         ((partialkey[31] & 0xFF)      );
      maxcounter = (1 << n) - 1;
```

Now comes the computational heart of the program. We set up a parallel team, which executes a parallel region, which in turn executes a parallel for loop with the index going from 0 to $2^n - 1$ (`maxcounter`), inclusive.

```
// Do trial encryptions in parallel.
new ParallelTeam().execute (new ParallelRegion()
   {
   public void run() throws Exception
      {
      execute (0, maxcounter, new IntegerForLoop()
         {
```

We declare the per-thread variables as instance fields of the parallel for loop subclass.

```
         // Thread local variables.
         byte[] trialkey;
         byte[] trialciphertext;
         AES256Cipher cipher;
```

We initialize the per-thread variables in the parallel for loop's `start()` method.

```
         // Set up thread local variables.
         public void start()
            {
            trialkey = new byte [32];
            System.arraycopy
               (partialkey, 0, trialkey, 0, 32);
            trialciphertext = new byte [16];
            cipher = new AES256Cipher (trialkey);
            }
```

The sequential program's loop body ends up in the parallel for loop's `run()` method.

```
         // Try every possible combination of low-order key
         // bits.
         public void run (int first, int last)
            {
            for (int counter = first; counter <= last;
                     ++ counter)
               {
               // Fill in low-order key bits.
               int lsbs = keylsbs | counter;
               trialkey[28] = (byte) (lsbs >>> 24);
```

```
                      trialkey[29] = (byte) (lsbs >>> 16);
                      trialkey[30] = (byte) (lsbs >>>  8);
                      trialkey[31] = (byte) (lsbs       );

                      // Try the key.
                      cipher.setKey (trialkey);
                      cipher.encrypt (plaintext, trialciphertext);

                      // If the result equals the ciphertext, we
                      // found the key.
                      if (match (ciphertext, trialciphertext))
                         {
```

If a thread finds the correct key, it copies the key into a temporary byte array and writes the array reference into the shared `foundkey` variable. The JVM synchronizes multiple threads writing `foundkey`, preventing write-write conflicts.

```
                         byte[] key = new byte [32];
                         System.arraycopy (trialkey, 0, key, 0, 32);
                         foundkey = key;
                         }
                      }
                   }
                });
             }
          });
```

At the conclusion of the parallel computation, the main program prints the results and exits as before.

```
       // Stop timing.
       long t2 = System.currentTimeMillis();

       // Print the key we found.
       System.out.println (Hex.toString (foundkey));
       System.out.println ((t2-t1) + " msec");
       }

    /**
     * Returns true if the two byte arrays match.
     */
    private static boolean match
       (byte[] a,
        byte[] b)
```

```
      {
      boolean matchsofar = true;
      int n = a.length;
      for (int i = 0; i < n; ++ i)
         {
         matchsofar = matchsofar && a[i] == b[i];
         }
      return matchsofar;
      }
   }
```

And that's it! To go from the sequential program to the SMP parallel program, we moved some of the variables' declarations to a different point to make them per-thread variables. We added a parallel team, parallel region, and parallel for loop. We moved the per-thread variable initialization into the parallel for loop's `start()` method. We moved the loop body into the parallel for loop's `run()` method. And we changed the code that saves the correct key to prevent thread conflicts. The parallel program's structure is much the same as the sequential program's, except the parallel computation code follows the nested parallel team–parallel region–parallel for loop idiom we studied in Chapter 6.

We will put off examining the FindKeySeq and FindKeySmp programs' running times until Chapter 8. Before moving on, though, let's consider a slight variation of these programs.

## 7.3 Early Loop Exit

As currently written, the AES key search programs try all possible values for the missing key bits, even though there's no need to continue once the correct key is found. Let's change that. In the sequential FindKeySeq program, when the correct key is found, it's easy enough to do an **early loop exit** by adding a break statement.

```
   for (int counter = 0; counter < maxcounter; ++ counter)
      {
      . . .
      if (match (ciphertext, trialciphertext))
         {
         . . .
         break;
         }
      }
```

However, if we add a similar break statement to the parallel FindKeySmp program, it won't work. The break statement will exit the loop in the parallel team thread that happens to find the correct key, but the other threads will stay in their loops trying useless keys. What we really want is for *all* the threads to exit their loops as soon as *any* thread finds the key.

Here's one way to do it. At the top of the loop, test `foundkey` and exit the loop if it is not null. This works because `foundkey` is initially null, and when the correct key is discovered, `foundkey` is set to a non-null array reference. Because `foundkey` is a *shared* variable (declared as a static field of the main program class), all the threads will be testing and setting the same variable, hence all the threads will exit their loops as soon as any thread finds the key. Reads and writes of an array reference variable are atomic, so no additional synchronization is needed when testing `foundkey`.

```
new ParallelTeam().execute (new ParallelRegion()
   {
   public void run() throws Exception
      {
      execute (0, maxcounter, new IntegerForLoop()
         {
         . . .
         public void run (int first, int last)
            {
            for (int counter = first;
                     counter <= last && foundkey == null;
                     ++ counter)
               {
               . . .
               if (match (ciphertext, trialciphertext))
                  {
                  byte[] key = new byte [32];
                  System.arraycopy (trialkey, 0, key, 0, 32);
                  foundkey = key;
                  }
               }
            }
         });
      }
   });
```

The FindKeySeq2 and FindKeySmp2 programs in the Parallel Java Library are sequential and SMP parallel versions of the AES key search program with an early loop exit as soon as the correct key is found. They will usually finish sooner than the original versions.

It's high time we measured and analyzed the SMP parallel AES partial key search program's performance. That will be the subject of Chapters 8, 9, and 10.

*This page intentionally left blank*

# CHAPTER 8

# Measuring Speedup

in which we define several metrics for a parallel program's performance; we predict what the ideal theoretical metrics should look like; and we measure the actual metrics for our parallel program

# CHAPTER 8 Measuring Speedup

## 8.1 Speedup Metrics

In Chapter 1, we said that one way parallel computing could help with today's massive computational problems is to reduce the time needed to get the results—a **speedup**. The time has come to define speedup precisely and to see how much speedup a real parallel program can achieve.

A program's **problem size**, $N$, is the number of computations the program performs to solve that problem. The particular problem determines how the problem size is measured. For an image-processing problem, the problem size might be the number of pixels in the image. For an $n \times n$-pixel image, the problem size would be $N = n^2$. For the AES key search problem, the problem size is the number of keys tested: $N = 2^n$, where $n$ is the number of missing key bits. In general, the problem size is defined so that the amount of computation is proportional to $N$.

A program's **running time**, $T$, is the amount of time the program takes to compute the answer to a problem. Many factors influence $T$. The computer's hardware characteristics—such as CPU clock speed, memory speed, caches, and so on—affect $T$; the faster the computer, the shorter the running time. $T$ depends on the problem size; the larger the problem, the longer the running time. Furthermore, the algorithm used to solve the problem determines how quickly $T$ increases as $n$ increases; an $O(n \log n)$ algorithm's running time will not grow nearly as quickly as an $O(n^2)$ algorithm's. $T$ also depends on how the program is implemented; the same algorithm can sometimes run faster when coded differently. $T$ depends on the number of processors $K$; adding more processors reduces the running time (one hopes).

When comparing the performance of different versions of a program, such as sequential and parallel versions using the same basic algorithm, we will always run the programs on the same computer. Furthermore, each processor of the parallel computer will have the same hardware characteristics; each processor will have the same CPU clock speed, for example. Thus, there will be no variation in the running times due to different algorithms or different hardware characteristics, and the only factors that influence $T$ are $N$ and $K$. We use the notation $T(N,K)$ to emphasize that the running time is a function of the problem size and the number of processors. When we need to distinguish the running times of the sequential version and the parallel version of a certain program, we write $T_{\text{seq}}(N,K)$ and $T_{\text{par}}(N,K)$.

A program's **speed**, $S$, is the rate at which program runs can be done. Speed is the reciprocal of running time:

$$S(N,K) = \frac{1}{T(N,K)} \tag{8.1}$$

$T$ is measured in seconds per program run, $S$ is measured in program runs per second.

A program's **speedup** is the speed of the *parallel* version running on *K* processors relative to the speed of the *sequential* version running on one processor for a given problem size *N*:

$$Speedup(N,K) = \frac{S_{\text{par}}(N,K)}{S_{\text{seq}}(N,1)} \tag{8.2}$$

Note that the denominator is $S_{\text{seq}}(N,1)$, not $S_{\text{par}}(N,1)$. Speedup compares the parallel version of a program to the sequential version, not the parallel version to itself. If we were solving the problem on one processor, we would run the sequential version of the program to avoid the parallel version's extra overhead. (Unfortunately, some published papers calculate the speedup of the parallel version with respect to itself rather than the sequential version, which can result in misleading performance numbers.)

Substituting Equation 8.1 into Equation 8.2 yields a formula for calculating speedup directly from running time, which is more convenient:

$$Speedup(N,K) = \frac{\dfrac{1}{T_{\text{par}}(N,K)}}{\dfrac{1}{T_{\text{seq}}(N,K)}} = \frac{T_{\text{seq}}(N,K)}{T_{\text{par}}(N,K)} \tag{8.3}$$

Ideally, a parallel program should run twice as fast on two processors as on one processor, three times as fast on three processors, four times as fast on four processors, and so on. Thus, ideally, *Speedup* should equal *K*. A plot of the ideal *Speedup* versus *K* is a straight line with unity slope, so this ideal speedup is also called a **linear speedup**.

As we will see, however, real parallel programs usually fall short of this ideal. **Efficiency** is a metric that captures how close to ideal a program's speedup is:

$$Eff(N,K) = \frac{Speedup(N,K)}{K} \tag{8.4}$$

An ideal parallel program has an efficiency of 1 for all problem sizes *N* and all numbers of processors *K*. A real parallel program typically has an efficiency less than 1, so that the speedup is less than *K*—a **sublinear speedup**. When designing parallel programs, we nonetheless strive to achieve the goal of a linear speedup.

The only way to know how closely we have achieved our goal is to run the program with a range of *N* and *K* values, measure *T*, and calculate the speedup and efficiency. However, it helps if we know what the speedup and efficiency are "supposed" to look like as a function of *N* and *K*. If the speedup and efficiency measurements don't look the way they're supposed to, the program's design might have a problem that must be fixed.

## 8.2 Amdahl's Law

Forty years ago, Gene Amdahl—the chief designer of IBM's System/360 family of mainframe computers, and who later founded his own company, Amdahl Corporation, to make IBM plug-compatible mainframes—published a short paper titled "Validity of the single processor approach to achieving large scale computing capabilities." In this paper, he compared two approaches for solving massive computational problems—the single-processor approach (sequential programs) and the then-newfangled multiple-processor approach (parallel programs). Amdahl's key insight was that a certain portion of any program must be executed sequentially—nowadays, we would say in a single thread. This portion consists of initialization, cleanup, thread synchronization, and similar housekeeping overhead, as well as I/O in some programs. This sequential portion can use only one processor, no matter how many processors are available in the parallel computer. Consequently, there is an upper bound on the speedup a parallel program can achieve.

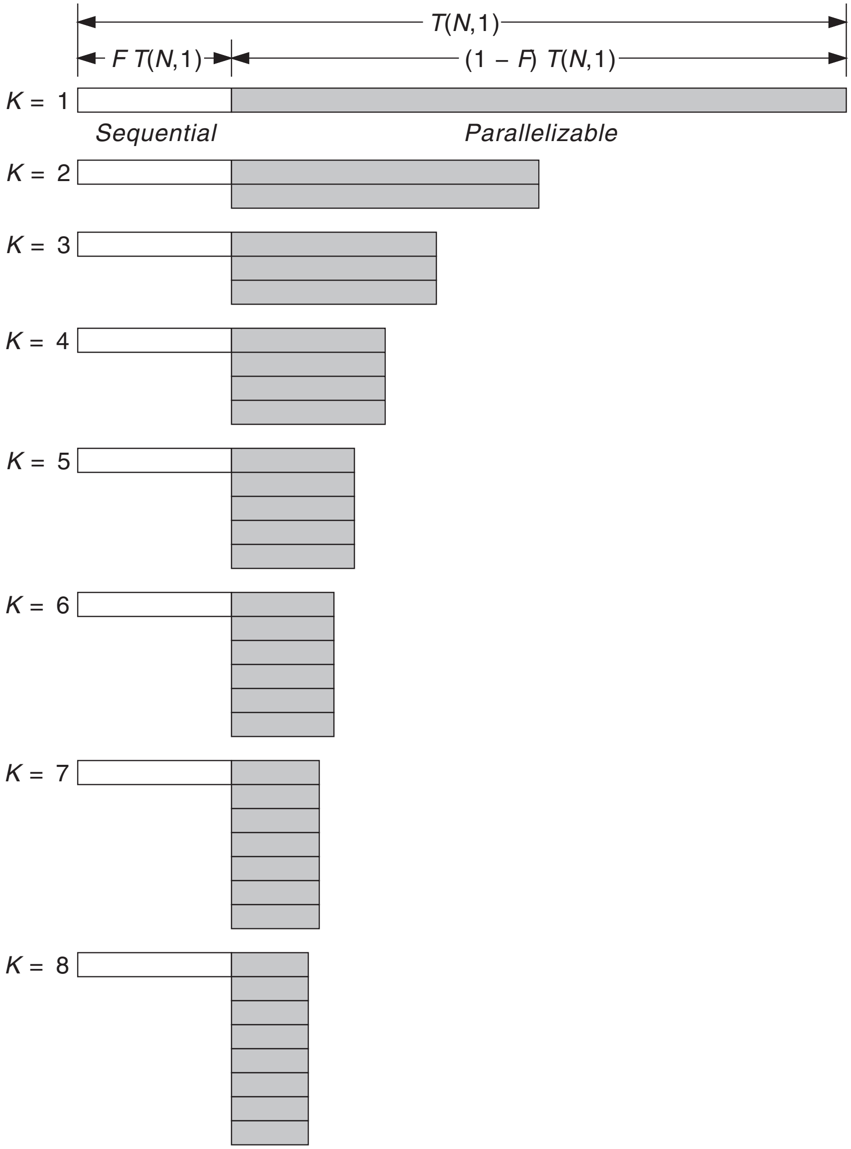


**Figure 8.1** A parallel program with running time $T(N,1)$ and sequential fraction $F$ executing with different numbers of processors $K$

Let the **sequential fraction** *F*, where $0 \le F \le 1$, be the fraction of a program that must be executed sequentially. If, for a given problem size *N*, a program's running time on a single processor is *T*(*N*,1), then $F \cdot T(N,1)$ of the running time must be executed sequentially and $(1–F) \cdot T(N,1)$ can be executed in parallel. If the latter portion of the running time is split equally among *K* processors (Figure 8.1), then the parallel program's running time is:

$$T(N,K) = F \cdot T(N,1) + \frac{1}{K}(1-F) \cdot T(N,1) \tag{8.5}$$

Equation 8.5 is known as **Amdahl's Law**, although that moniker was bestowed by others, not by Amdahl.

From Amdahl's Law, we can derive equations for speedup and efficiency as a function of the sequential fraction:

$$\begin{aligned} \mathit{Speedup}(N,K) &= \frac{T(N,1)}{T(N,K)} \\ &= \frac{T(N,1)}{F \cdot T(N,1) + \frac{1}{K}(1-F) \cdot T(N,1)} \\ &= \frac{1}{F + \frac{1-F}{K}} \end{aligned} \tag{8.6}$$

$$\mathit{Eff}(N,K) = \frac{\mathit{Speedup}(N,K)}{K} = \frac{1}{KF + 1 - F} \tag{8.7}$$

Consider what happens to the speedup as the number of processors increases. In the limit as *K* goes to infinity, the speedup (Equation 8.6) goes to 1/*F*. No matter how many processors we add, we will never achieve a speedup greater than the reciprocal of the program's sequential fraction. Furthermore, in the limit as *K* goes to infinity, the efficiency (Equation 8.7) goes to 0. As we add processors, the efficiency just gets worse and worse. Figures 8.2 and 8.3 plot speedup and efficiency from Amdahl's Law as a function of *K* for several values of *F*.

From Amdahl's Law, we can gain three important insights regarding parallel program design. The first insight is that a program's sequential fraction *F* has to be very small if we want to achieve good speedup and efficiency as we scale up the number of processors in our parallel computer. For example, Figure 8.3 shows that if we want an efficiency of 90 percent or better as we scale up to 100 processors, then we need a sequential fraction of 0.001 or less. In other words, no more than one-tenth of one percent of the running time is allowed to execute on a single processor, and all the rest has to execute in parallel. Sequential overhead in a parallel program severely reduces the program's performance, and when designing parallel programs, we go to great lengths, if necessary, to reduce *F*.

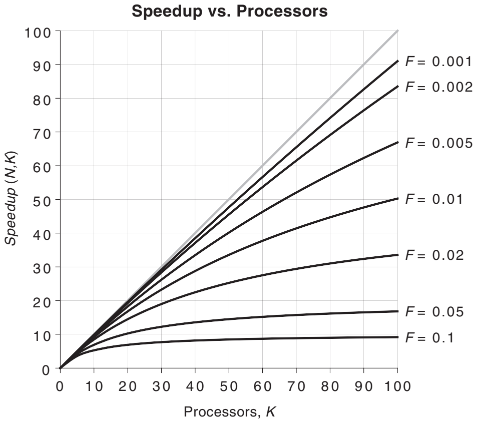


**Figure 8.2** Speedup predicted by Amdahl's Law

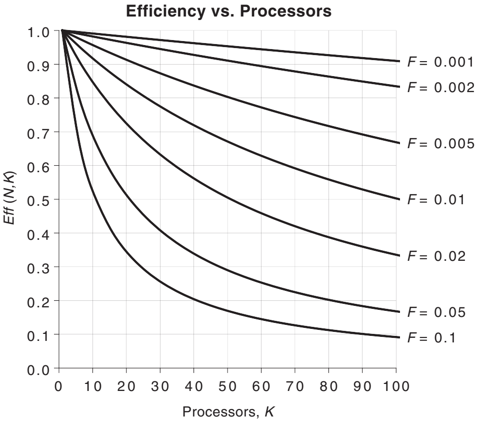


**Figure 8.3** Efficiency predicted by Amdahl's Law

However, there is another way to get good performance as the number of processors scales up. Some 20 years after Amdahl's paper John Gustafson published a short paper titled "Reevaluating Amdahl's Law," in which he pointed out that if the problem size is also scaled up as the number of processors is scaled up—what we call a **sizeup**—the program can achieve a near-linear speedup, seemingly contradicting Amdahl's Law. We defer further discussion of Gustafson's observation until Chapter 10. For now, we simply note that Amdahl's Law is not the last word in parallel program performance.

The second insight from Amdahl's Law is the expected shape of the plot of a program's efficiency versus the number of processors. Figure 8.4 plots the efficiencies we expect to see when running a parallel program on an eight-processor SMP parallel computer. The measured efficiencies should start near 1 and decrease in roughly a straight line as *K* increases, unless *F* is relatively large. If the measured efficiency plots don't resemble Figure 8.4, we must figure out what's going on and change the program's design to fix the problem.

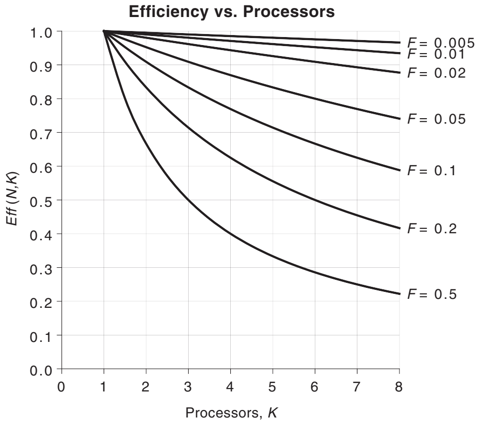


**Figure 8.4** Efficiency for an 8-processor SMP parallel computer

Suppose we have measured a program's running time as a function of problem size and number of processors, *T*(*N*,*K*), and have plotted the program's efficiency. The plot looks more or less like Figure 8.4, with efficiency decreasing as *K* increases. But how can we tell whether the efficiency curves are behaving as Amdahl's Law predicts? The third insight from Amdahl's Law yields another metric we can use to analyze a program's performance. Rearranging Equation 8.5 gives a formula for *F* as a function of *T* and *K*:

$$F = \frac{K \cdot T(N,K) - T(N,1)}{K \cdot T(N,1) - T(N,1)} \tag{8.8}$$

Equation 8.8 lets us calculate *F* from the running-time measurements, namely *T*(*N*,1) and *T*(*N*,*K*), for the parallel program. When *F* is determined from the data in this way, it is called the **experimentally determined sequential fraction**, *EDSF*. (Note that we can only calculate *EDSF* for $K \geq 2$.)

In a 1990 paper titled "Measuring parallel processor performance," Alan Karp and Horace Flatt pointed out how the *EDSF* metric can help diagnose problems in a parallel program's performance. If the program is behaving as shown in Figure 8.1—a sequential portion with a fixed running time plus a parallel portion with a running time inversely proportional to *K*—then *F* should be a constant, and a plot of *EDSF* versus *K* should be a horizontal line. If the measured *EDSF* plot does not show up as roughly a horizontal line, it's another indication that something is going on that might require changing the program's design.

Armed with these insights, we are ready to start measuring and analyzing the AES key search program's speedup, efficiency, and *EDSF*.

## 8.3 Measuring Running Time

The AES key search program measures its own wall-clock running time using the `System.currentTimeMillis()` method. However, two program runs hardly ever yield the same running time measurements, even with identical inputs. There are several reasons for this. The system clock is not perfectly accurate; readings may vary a few tens of milliseconds on typical systems. Chiefly, however, other user programs and background processes running on the same computer take CPU time away from the parallel program, increasing the parallel program's wall-clock running time by an unpredictable amount on each run. We need a way to get a meaningful running time measurement for the program despite all these random fluctuations.

Faced with random errors in an experimental measurement, it's likely that your first thought is to take several readings and use their average. However, this is *not* the proper procedure for measuring a program's running time. To see why, we have to examine the often-unstated assumption behind the procedure of averaging a series of measurements.

An experimental reading of some quantity consists of the true value, known only to Mother Nature, plus a **measurement error** arising from the measuring apparatus. Mathematically, the measurement error is modeled as a random variable with some probability distribution. If we assume that *the measurement error obeys a Gaussian probability distribution with a mean of zero*, then taking the average of a series of measurements gives a close approximation to the true value. Figure 8.5 illustrates why. In this example, the true value of the running time being measured is $T = 10{,}000$ msec. If the measurement error obeys a Gaussian distribution with a mean of zero and a certain variance $\sigma^2$, the measurements obey a Gaussian distribution with a mean of 10,000 and the same variance (the bell-shaped curve). The black dots are several measurements, which are samples of the Gaussian distribution. The **sample mean** $\overline{T}$ is the average of the measurements. Probability theory tells us that with $n$ measurements, $\overline{T}$ obeys a Gaussian distribution with the same mean as $T$, 10,000, but with a variance of only $\sigma^2/n$. Thus, $\overline{T}$ fluctuates around the true value less widely than the raw measurements, making $\overline{T}$ more likely to be closer to, hence a better estimator of, the true value than any of the raw measurements. In Figure 8.5, even though the measured $T$ values fluctuate between about 5,000 and 17,000, the sample mean of 10,095.2 ends up quite close to the true $T$ value.

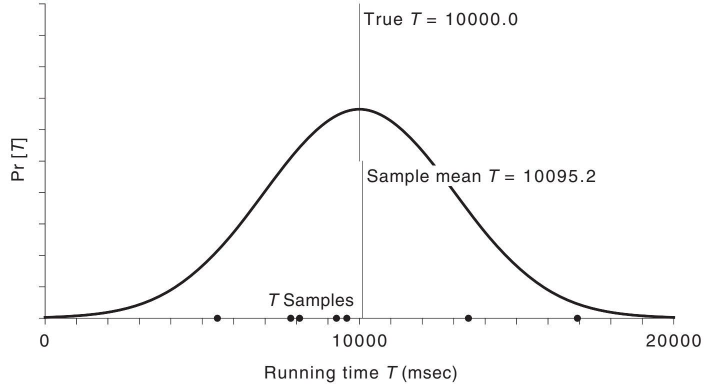


**Figure 8.5** Running-time measurement with a Gaussian error distribution

The problem is that when measuring a program's wall-clock running time, *the measurement error distribution is not a zero-mean Gaussian.* A Gaussian distribution is symmetric; positive and negative errors both are equally possible. But the chief source of errors in the program's wall-clock running time does not have a symmetric distribution. When the operating system takes the CPU away from the parallel program to respond to an interrupt or to let another process run, the parallel program's running-time measurement can only increase, never decrease. The measurement-error probability distribution ends up looking more like Figure 8.6, which shows that the sample mean always ends up being larger than the true value. When the measurement error is always positive, the best estimator of the true value is the *minimum* of the measurements, not the average.

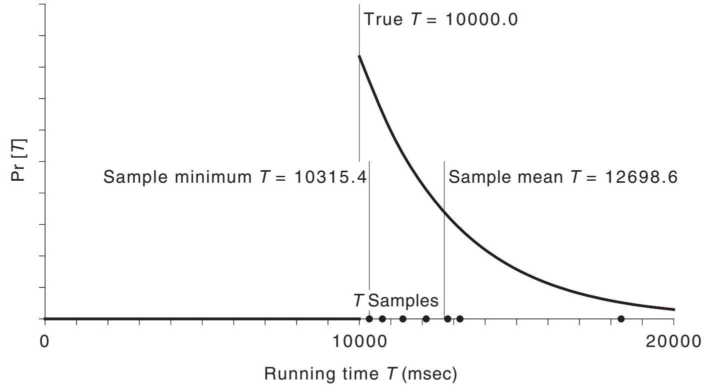


**Figure 8.6** Running time measurement with a positive error distribution

Based on these considerations, here's the procedure we use in this book to measure our programs' running times:

1. Ensure that the parallel program is the only user process running. Don't let other users log in while timing measurements are being made. Don't run other programs such as editors, Web browsers, or e-mail clients.
2. To the extent possible, don't have any server or daemon processes running, such as Web servers, e-mail servers, file servers, network time daemons, and so on.
3. Prepare several input data sets, covering a range of problem sizes $N$. Choose the smallest problem size so that $T_{\text{seq}}(N,1)$ is at least 60 seconds.
4. For each $N$ (each input data set), run the sequential version of the program seven times. (The number of runs, seven, is just an arbitrary choice.) Take the smallest of the running times as the measured $T_{\text{seq}}(N,1)$ value.
5. For each $N$, and for each $K$ from 1 up to the number of available processors, run the parallel version of the program seven times. Take the smallest of the running times as the measured $T_{\text{par}}(N,K)$ value.

There are two reasons for measuring only problem sizes whose running times are 60 seconds or more. First, if we're willing to wait a certain amount of time for the program to finish before losing patience, and the program takes less time than this "impatience threshold" even on one processor, then there's no point in trying to reduce the running time by going to multiple processors. For this book, we arbitrarily peg the impatience threshold at 60 seconds. Second, when a Java program starts, the JVM needs some time to "warm up"—to load and verify the class files, to detect hot spots, and to run the JIT compiler to compile the hot spots' bytecode to machine instructions. This warm-up overhead contributes to the program's sequential portion. Running the program for a longer time reduces the sequential fraction due to JVM warm-up.

## 8.4 FindKeySmp Running Time Measurements

Table 8.1 (at the end of the chapter) gives the running-time measurements in milliseconds for the AES key search program, as well as the speedups, efficiencies, and *EDSFs* calculated from the running times. The programs were run on a computer named "parasite," a Sun Microsystems eight-processor SMP parallel computer with four UltraSPARC-IV dual-core CPU chips, a 1.35 GHz CPU clock speed, and 16 GB of main memory. The six input data sets had from $n = 24$ to 29 missing key bits, yielding problem sizes of $N = 16\text{M}$, 32M, 64M, 128M, 256M, and 512M encryption keys tested. ("M" stands for $2^{20}$.) In the $K$ column, "seq" denotes the sequential version of the program (FindKeySeq) and 1 through 8 denote the parallel version of the program (FindKeySmp). The $T$ column lists the smallest running time among the seven program runs.

Figure 8.7 plots running time versus number of processors for the AES key search program. Note that this is a **log-log plot**—both axes use a logarithmic scale rather than a linear scale. (See Appendix C for further information about log-log plots.) The logarithmic scale is better suited to plotting data that spans many orders of magnitude, as does the running time data. Furthermore, the log-log plot lets us eyeball whether the parallel program is achieving ideal performance. Ideally, $T$ should be proportional to $K^{-1}$. This would show up on a log-log plot as a straight line with a slope of $-1$. If the $T$ versus $K$ curves don't look like that, the program's performance is not ideal. As we can see, the AES key search program's performance is less than ideal, especially for larger numbers of processors.

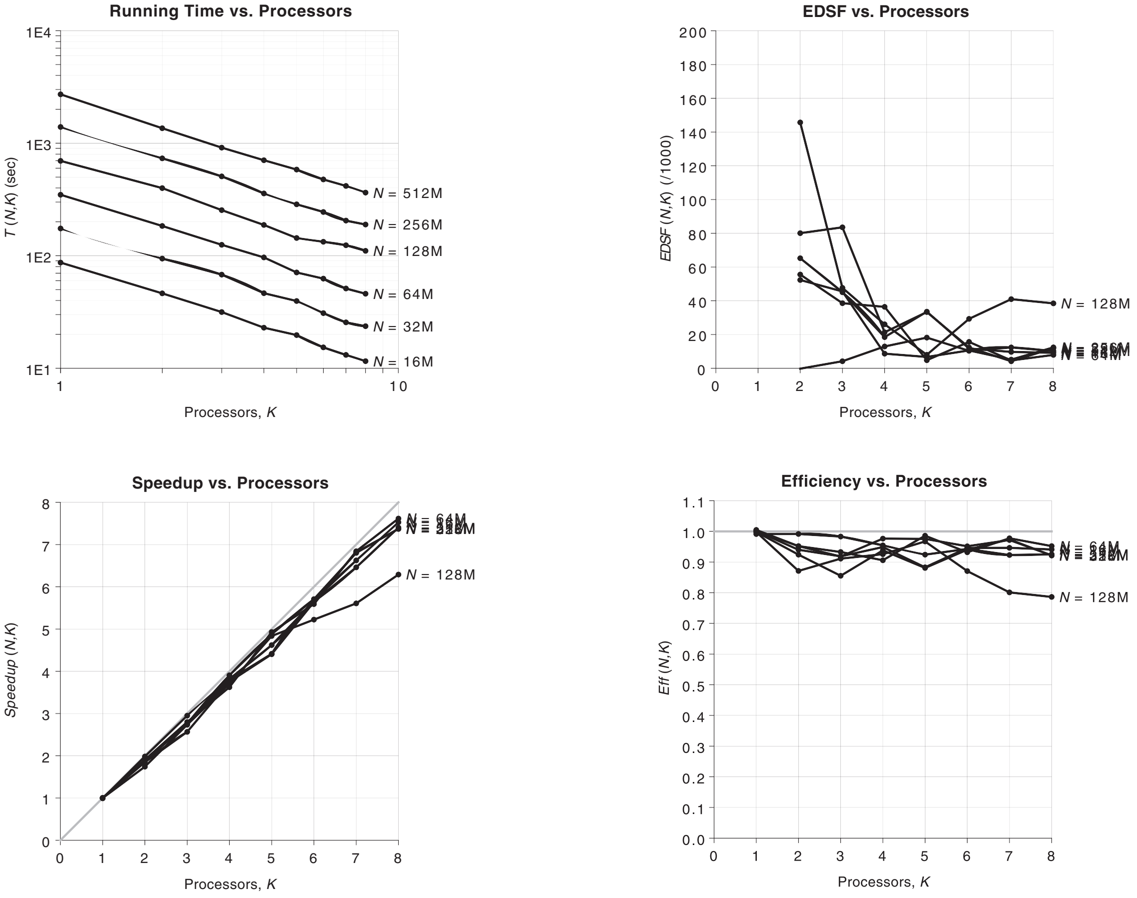


**Figure 8.7** FindKeySeq/FindKeySmp running-time metrics

Figure 8.7 also plots speedup, efficiency, and *EDSF* versus number of processors for the AES key search program. These curves reinforce what we saw in the running-time curves. For smaller numbers of processors, we are getting close to linear speedups and efficiencies close to 1, but for larger numbers of processors, the speedup and efficiency curves start jumping around. The efficiency curves in particular do not look anything like what we would expect from Figure 8.4. The *EDSF* curves are all over the map, and are nowhere near constant. Something is going on in the parallel program that must be fixed. This will be the topic of Chapter 9.

By the way, the running-time metric plots in Figure 8.7 were produced by a Java program, class Speedup, in the Parallel Java Library. The program takes the raw running-time measurements as in Table 8.1, calculates and prints the running time, speedup, efficiency, and *EDSF* metrics, and generates the plots. On each plot, the program labels each curve with the corresponding problem size ($N$ = 16M, $N$ = 32M, and so on). If the curves fall on top of each other—as often happens with the speedup and efficiency curves—then the labels fall on top of each other as well. Refer to the Parallel Java documentation for instructions on how to use the Speedup program.

## 8.5 For Further Information

Amdahl's original paper on parallel program performance:

- G. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the AFIPS Spring Joint Computer Conference*, 1967, pages 483–485.

Gustafson's original paper on parallel program performance:

- J. Gustafson. Reevaluating Amdahl's law. *Communications of the ACM*, 31(5):532–533, May 1988.

A critique of Amdahl's and Gustafson's work:

- Y. Shi. Reevaluating Amdahl's law and Gustafson's law. October 1996. http://joda.cis.temple.edu/~shi/docs/amdahl/amdahl.html

Karp's and Flatt's original paper on parallel program performance:

- A. Karp and H. Flatt. Measuring parallel processor performance. *Communications of the ACM*, 33(5):539–543, May 1990.

**Table 8.1** FindKeySeq/FindKeySmp running-time metrics

| *N* | *K* | *T* | *Spdup* | *Eff* | *EDSF* | *N* | *K* | *T* | *Spdup* | *Eff* | *EDSF* |
|---:|---:|---:|---:|---:|---:|---:|---:|---:|---:|---:|---:|
| 16M | seq | 87026 | | | | 128M | seq | 694908 | | | |
| 16M | 1 | 86914 | 1.001 | 1.001 | | 128M | 1 | 696352 | 0.998 | 0.998 | |
| 16M | 2 | 46291 | 1.880 | 0.940 | 0.065 | 128M | 2 | 398902 | 1.742 | 0.871 | 0.146 |
| 16M | 3 | 31581 | 2.756 | 0.919 | 0.045 | 128M | 3 | 254240 | 2.733 | 0.911 | 0.048 |
| 16M | 4 | 22931 | 3.795 | 0.949 | 0.018 | 128M | 4 | 187697 | 3.702 | 0.926 | 0.026 |
| 16M | 5 | 19715 | 4.414 | 0.883 | 0.034 | 128M | 5 | 143717 | 4.835 | 0.967 | 0.008 |
| 16M | 6 | 15333 | 5.676 | 0.946 | 0.012 | 128M | 6 | 133041 | 5.223 | 0.871 | 0.029 |
| 16M | 7 | 13138 | 6.624 | 0.946 | 0.010 | 128M | 7 | 123930 | 5.607 | 0.801 | 0.041 |
| 16M | 8 | 11561 | 7.528 | 0.941 | 0.009 | 128M | 8 | 110477 | 6.290 | 0.786 | 0.038 |
| 32M | seq | 174282 | | | | 256M | seq | 1396225 | | | |
| 32M | 1 | 174621 | 0.998 | 0.998 | | 256M | 1 | 1393962 | 1.002 | 1.002 | |
| 32M | 2 | 94301 | 1.848 | 0.924 | 0.080 | 256M | 2 | 733449 | 1.904 | 0.952 | 0.052 |
| 32M | 3 | 67931 | 2.566 | 0.855 | 0.084 | 256M | 3 | 506964 | 2.754 | 0.918 | 0.046 |
| 32M | 4 | 46449 | 3.752 | 0.938 | 0.021 | 256M | 4 | 357525 | 3.905 | 0.976 | 0.009 |
| 32M | 5 | 39578 | 4.404 | 0.881 | 0.033 | 256M | 5 | 286268 | 4.877 | 0.975 | 0.007 |
| 32M | 6 | 30895 | 5.641 | 0.940 | 0.012 | 256M | 6 | 244600 | 5.708 | 0.951 | 0.011 |
| 32M | 7 | 25567 | 6.817 | 0.974 | 0.004 | 256M | 7 | 205257 | 6.802 | 0.972 | 0.005 |
| 32M | 8 | 23649 | 7.370 | 0.921 | 0.012 | 256M | 8 | 189344 | 7.374 | 0.922 | 0.012 |
| 64M | seq | 349907 | | | | 512M | seq | 2693607 | | | |
| 64M | 1 | 348290 | 1.005 | 1.005 | | 512M | 1 | 2717345 | 0.991 | 0.991 | |
| 64M | 2 | 183820 | 1.904 | 0.952 | 0.056 | 512M | 2 | 1358338 | 1.983 | 0.992 | 0.000 |
| 64M | 3 | 125050 | 2.798 | 0.933 | 0.039 | 512M | 3 | 913330 | 2.949 | 0.983 | 0.004 |
| 64M | 4 | 96578 | 3.623 | 0.906 | 0.036 | 512M | 4 | 705661 | 3.817 | 0.954 | 0.013 |
| 64M | 5 | 70986 | 4.929 | 0.986 | 0.005 | 512M | 5 | 582974 | 4.620 | 0.924 | 0.018 |
| 64M | 6 | 62586 | 5.591 | 0.932 | 0.016 | 512M | 6 | 476279 | 5.656 | 0.943 | 0.010 |
| 64M | 7 | 51142 | 6.842 | 0.977 | 0.005 | 512M | 7 | 416946 | 6.460 | 0.923 | 0.012 |
| 64M | 8 | 45945 | 7.616 | 0.952 | 0.008 | 512M | 8 | 363923 | 7.402 | 0.925 | 0.010 |

# CHAPTER 9

# Cache Interference

in which we see how cache interference can ruin a parallel program’s performance; we study how to design programs to eliminate cache interference; and we measure the effect on our program’s running time

# CHAPTER 9 Cache Interference

## 9.1 Origin of Cache Interference

To understand why the SMP parallel program for AES key search from Chapter 7 performs as poorly as the metrics in Chapter 8 show, we have to take a detour into the internals of the JVM and the CPU.

Here again are the per-thread variables used by each thread in the FindKeySmp program's parallel team. These are declared as instance fields of the parallel for loop subclass and are initialized in the parallel for loop's `start()` method.

```
new ParallelTeam().execute (new ParallelRegion()
   {
   public void run() throws Exception
      {
      execute (0, maxcounter, new IntegerForLoop()
         {
         // Thread local variables.
         byte[] trialkey;
         byte[] trialciphertext;
         AES256Cipher cipher;

         // Set up thread local variables.
         public void start()
            {
            trialkey = new byte [32];
            System.arraycopy
               (partialkey, 0, trialkey, 0, 32);
                trialciphertext = new byte [16];
                cipher = new AES256Cipher (trialkey);
                }
```

When one of the parallel team threads creates an instance of the parallel for loop subclass, the JVM allocates a block of storage in the computer's main memory to hold the parallel for loop object's instance fields. The storage block consists of four bytes to store the reference to the `trialkey` array, four bytes to store the reference to the `trialciphertext` array, four bytes to store the reference to the cipher object, and a few extra bytes of overhead for the JVM's internal use. (This assumes a 32-bit CPU where memory addresses are 32 bits, or 4 bytes.)

When the `trialkey` variable is initialized in the parallel for loop's `start()` method with the expression `new byte [32]`, the JVM allocates another block of storage to hold the byte array and stores a reference to this block in the `trialkey` variable. The block consists of JVM overhead plus 32 bytes for the array elements. Likewise, the `trialciphertext` and `cipher` variables each get a block of storage. The `cipher` object is an instance of class AES256Cipher, so storage blocks are also allocated for the object itself and for its instance fields. When all the threads have finished initializing their per-thread variables, the storage blocks might end up arranged in main memory somewhat as shown in Figure 9.1. (Actually, the JVM is allowed to place each storage block wherever it wants; the JVM is not required to allocate one thread's storage blocks contiguously.)

However, the computer's CPUs do not access the main memory directly. As described in Chapter 2, a **cache memory** sits between the CPU and the main memory. When the CPU needs to read a memory location, it first reads the appropriate **cache line**, a block of contiguous memory locations that includes the desired location, from main memory into the cache. The CPU then reads the contents of the memory location from the cache. Subsequent reads of the same memory location, or any location in the same cache line, come from the fast cache rather than the slow main memory, thus speeding up the program's execution. The cache line size depends on the CPU hardware; it is typically 64 bytes or 128 bytes.

Figure 9.2 shows the cache line boundaries superimposed on the program variables' storage blocks. Note that while the cache line boundaries occur regularly every 64 or 128 memory locations, the boundaries between the variables' storage blocks can fall at any locations, depending on the sizes of the blocks. Therefore, it is quite possible for a particular cache line to contain pieces of different variables' storage blocks. In particular, it is possible for a particular cache line to contain pieces of *different threads' per-thread variables'* storage blocks—as is the case for the cache line in the middle of Figure 9.2.

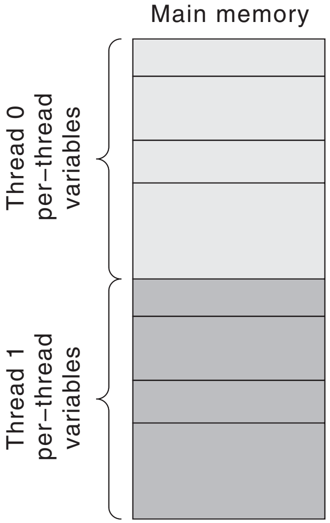


**Figure 9.1** Per-thread variable memory layout

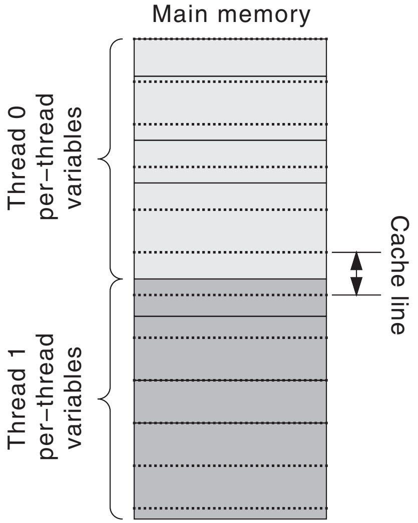


**Figure 9.2** Memory layout showing cache line boundaries

Recall from Chapter 2 that one example of a cache-management strategy involves a *write-through* policy for dirty cache lines and an *invalidation*-based cache-coherence protocol. Suppose one CPU has read a certain variable, so the variable's cache line has been loaded into that CPU's cache, and then that CPU writes a value into the variable. The cache's contents no longer agree with the main memory's contents, so the cache line is written from the cache back out to main memory. Furthermore, the writing CPU sends a signal telling the other CPUs to invalidate the cache line in question in the other CPUs' caches. This causes the other CPUs to read the cache line's new contents from the main memory the next time the other CPUs access the cache line. This is how a thread executing in one CPU obtains a new value stored into a shared variable by a thread executing in another CPU.

Now, consider what happens when one thread reads its own *per-thread* variable located in a certain cache line and another thread reads its own *per-thread* variable located in the same cache line. Each CPU loads the same cache line into its own cache, and then each CPU proceeds to read its own per-thread variable from the cache line. Suppose the first thread writes a new value into its own per-thread variable. The first thread's write causes the second thread's cache line to become invalidated. Suppose the second thread reads its own per-thread variable a second time. Normally, this variable would be read from the fast cache without needing to go to the main memory. But because the cache line was invalidated, the second thread's CPU has to reload the cache line from slow main memory before it can get the variable's value—even though the second thread's variable's value did not change. Thus, the overall program takes longer to run than it would if the threads never wrote the same cache lines. This effect is called **cache interference**.

Even though the threads never read or write the same *memory locations* (because each thread its accessing only its own per-thread variables), the threads do read and write the same *cache line* (because the JVM put different threads' per-thread variables at addresses that fell in the same cache line). The resulting performance reduction is similar to what happens if the threads have to take turns reading and writing a *shared* variable. But because the threads are not actually sharing the variables, the phenomenon is called **false sharing**.

Cache interference, then, is the reason the AES key search program fails to achieve the expected performance. The more threads (processors) there are, the more opportunities there are for each CPU to invalidate the other CPU's cache lines, and the greater the detriment to the program's performance.

## 9.2 Eliminating Cache Interference

To eliminate the cache interference that arises from false sharing of per-thread variables, we must ensure that different threads' per-thread variables never reside in the same cache line. One way to accomplish this would be for the JVM to allocate each storage block starting at a cache line boundary. However, JVM implementations don't do this because of all the wasted space that would result in the cache lines at the ends of the allocated blocks. A better way to avoid cache interference would be for the JVM to allocate the storage blocks for each thread's per-thread variables in separate regions of memory, with no overlapping cache lines. However, the Java language has no way for the programmer to declare variables as per-thread variables for purposes of memory allocation. (While the Java platform does provide class java.lang.ThreadLocal, this class does not cause variables to be allocated in different cache lines.)

Another way to eliminate cache interference is to add some **padding** in the memory layout. Suppose every per-thread variable's storage block has some extra bytes at the end, enough bytes to spill across the next cache line boundary. Suppose the program never reads or writes these extra padding bytes. Figure 9.3 shows the result. In Figure 9.3, the accessed portion of each variable's storage block is shaded, and the padding (non-accessed) portion is white. You can see that the shaded portion of one variable is never in the same cache line as the shaded portion of another variable. Consequently, cache interference does not occur.

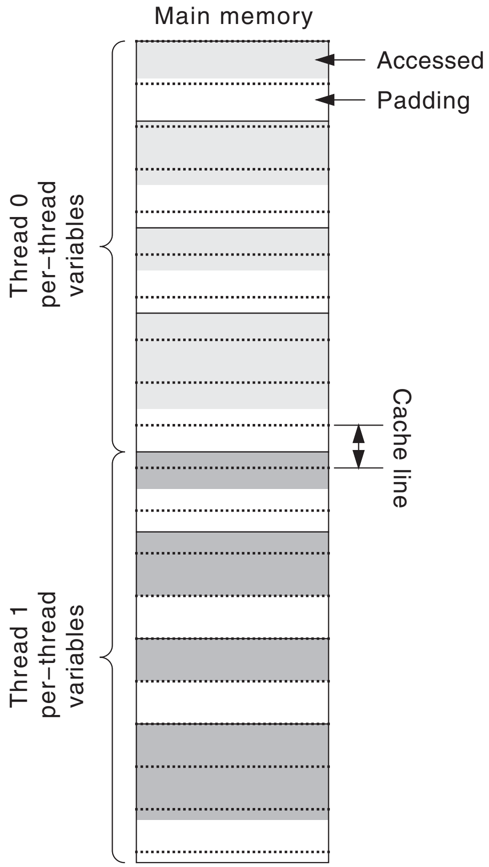


**Figure 9.3** Memory layout with extra padding

Here's how to get that extra padding. First, when declaring a class's instance fields, throw in some extra padding fields. Not knowing what the cache line size will be when we run the program, we'll be conservative and add 128 bytes of padding. Sixteen fields of type `long`, each of which occupies eight bytes, does the job. Here are the AES key search program's parallel for loop subclass's instance fields, with padding.

```
    // Thread local variables.
   byte[] trialkey;
   byte[] trialciphertext;
   AES256Cipher cipher;
```

```
// Extra padding.
long p0, p1, p2, p3, p4, p5, p6, p7;
long p8, p9, pa, pb, pc, pd, pe, pf;
```

Note that getting the padding by declaring a 16-element `long` array instance field (or a 128-element `byte` array instance field) *does not work*. The problem is that only a four-byte *reference* to the array ends up in the parallel for loop subclass's storage block; the actual array elements end up in a separate block. The padding fields must be of a primitive type, such as `long`, for them to be located in the parallel for loop subclass's storage block.

Second, when creating a new array, allocate enough extra array elements to occupy 128 bytes. Here is the parallel for loop subclass's `start()` method, allocating extra padding in the byte arrays.

```
// Set up thread local variables.
public void start()
    {
    trialkey = new byte [32+128]; // + padding
    System.arraycopy
        (partialkey, 0, trialkey, 0, 32);
    trialciphertext = new byte [16+128]; // + padding
    cipher = new AES256CipherSmp (trialkey);
    }
```

The `start()` method also creates an instance of class AES256CipherSmp instead of class AES256Cipher. Class AES256CipherSmp has extra padding in *its* instance fields. To eliminate cache interference, *all* the per-thread storage blocks must include the extra padding.

This technique of adding padding bytes is not a perfect solution to the problem of cache interference. It increases the amount of storage the program consumes. It requires either knowing the machine's cache line size (which reduces the program's portability) or picking an amount of padding one hopes is larger than any machine's cache line size (which may increase storage usage unnecessarily). But most of all, it requires that the programmer add code to do something the Java compiler or JVM should do. Perhaps a future version of the Java platform will automatically place per-thread variables in memory so as to avoid false sharing.

## 9.3 FindKeySmp3 Measurements

Program FindKeySmp3 in the Parallel Java Library is the same as program FindKeySmp from Chapter 7, except it incorporates padding to avert cache interference, as described earlier. To see whether the padding made a difference, we measure the FindKeySmp3 program's running time on the same "parasite" SMP parallel computer with the same input data sets as the FindKeySmp program. As it happens, the "parasite" computer's cache line size is 128 bytes.

Table 9.1 (at the end of the chapter) gives the running-time measurements in milliseconds for the modified AES key search program for various problem sizes *N*, as well as the speedups, efficiencies, and *EDSFs* calculated from the running times. Figure 9.4 plots the running-time metrics versus the number of processors. The *T* versus *K* plot looks a lot better than the FindKeySmp program's plot—it has nice straight lines. The *Speedup* versus *K* and *Eff* versus *K* plots also look dramatically better, just as they should according to Amdahl's Law, with efficiencies greater than 0.92 for all problem sizes. Adding extra padding in the memory layout has definitely eliminated the effects of cache interference.

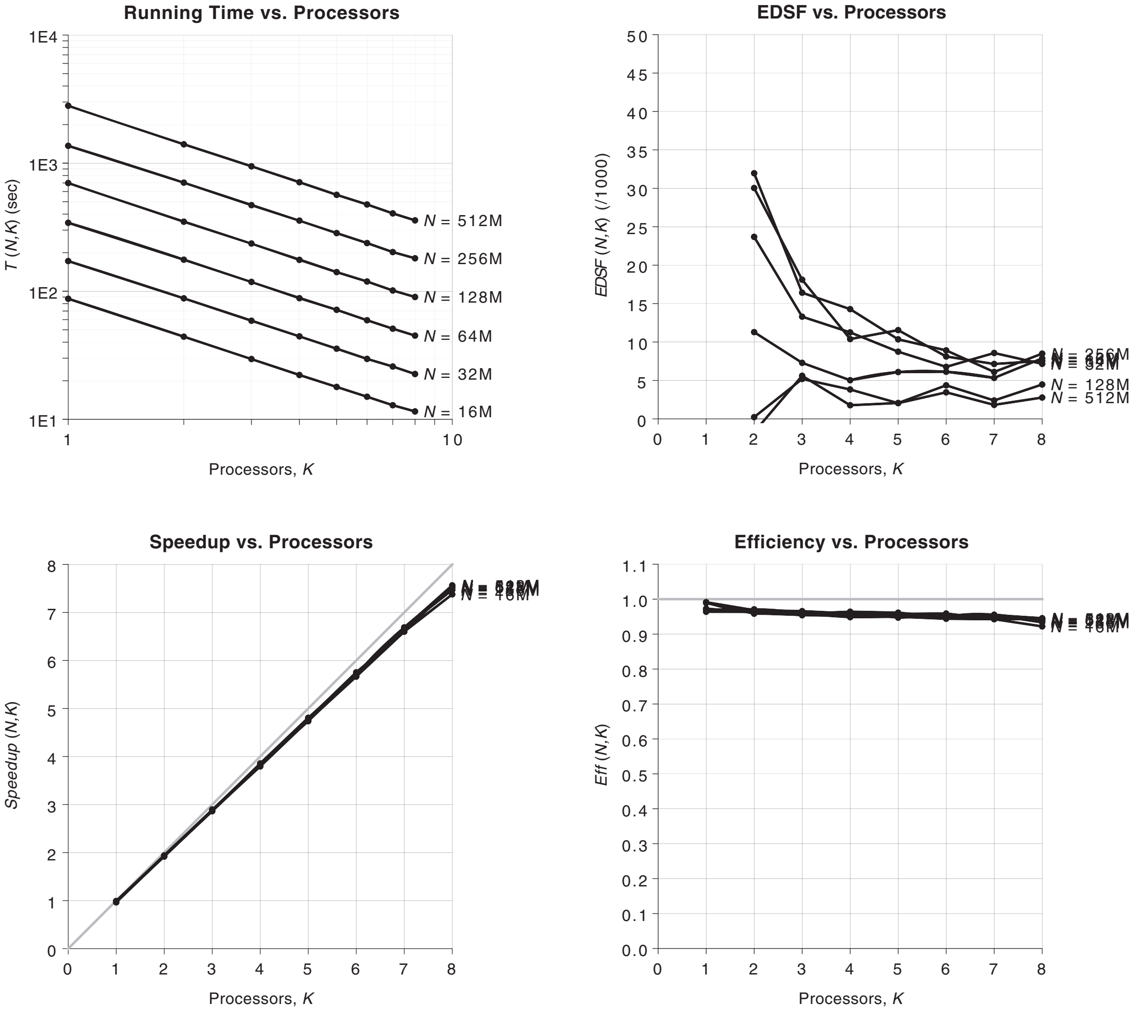
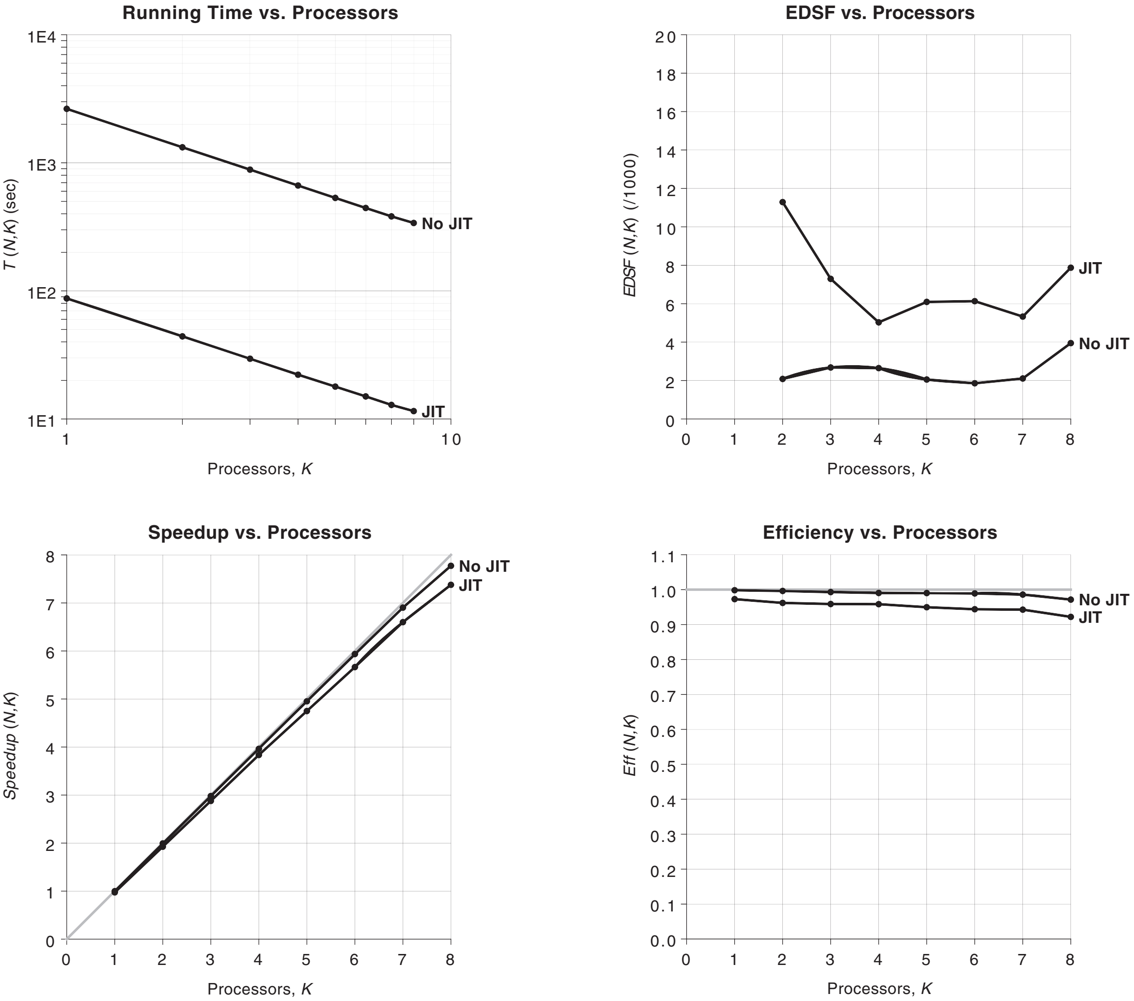


**Figure 9.4** FindKeySeq/FindKeySmp3 running-time metrics

The *EDSF* versus *K* plot still looks strange; the curves are not even close to being horizontal lines. The fluctuations we see are due to two things: random measurement error, and the JVM's JIT compiler.

To illustrate just how the JIT compiler affects a Parallel Java program's performance, Table 9.2 (at the end of the chapter) lists the running-time metrics for a problem size of $N = 16\text{M}$ with the JIT compiler disabled. Figure 9.5 compares the running-time metrics for $N = 16\text{M}$ with and without the JIT compiler. From the plots, it's apparent that the JIT compiler introduces a small but definite reduction in the speedup and efficiency. Without the JIT compiler, the *EDSF* plot is nearly constant, as it should be, with small fluctuations due to random measurement error. With the JIT compiler enabled, the sequential fraction increases, as does the variation in the *EDSF* curve. The JIT compiler reduces the speedup, reduces the efficiency, and increases the sequential fraction by taking a bit of CPU time away from the program's computations to compile the hot spots to machine code.


Running Time vs. Processors

EDSF vs. Processors

Speedup vs. Processors

Efficiency vs. Processors

**Figure 9.5** FindKeySeq/FindKeySmp3 running time metrics for $N = 16\text{M}$, with and without the JIT compiler

If the JIT compiler has negative effects on the speedup, efficiency, and sequential fraction, why not run with the JIT compiler disabled all the time? The plot of running time versus processors makes the answer clear. Without the JIT compiler, the program's running time increases by a factor of 30. The CPU can execute native machine code instructions 30 times faster than it can interpret Java bytecodes. A slight impact on the speedup, efficiency, and *EDSF* is well worth such a drastic reduction in the running time. We always run our Parallel Java programs with the JIT compiler enabled (which is the default).

Now that we've dealt with the cache interference in the AES key search program, in Chapter 10 we'll return to Gustafson's observation and look at what happens when we scale up the problem size as we scale up the number of parallel processors.

## 9.4 For Further Information

On cache-related and other issues affecting Java program performance on SMP parallel computers:

- Z. Cao, W. Huang, and J. Chang. A study of Java virtual machine scalability issues on SMP systems. In *Proceedings of the 2005 IEEE International Workload Characterization Symposium*, 2005, pages 119–128.

\

**Table 9.1** FindKeySeq/FindKeySmp3 running-time metrics

| *N* | *K* | *T* | *Spdup* | *Eff* | *EDSF* | *N* | *K* | *T* | *Spdup* | *Eff* | *EDSF* |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 16M | seq | 84981 | | | | 128M | seq | 677725 | | | |
| 16M | 1 | 87346 | 0.973 | 0.973 | | 128M | 1 | 699172 | 0.969 | 0.969 | |
| 16M | 2 | 44166 | 1.924 | 0.962 | 0.011 | 128M | 2 | 348984 | 1.942 | 0.971 | -0.002 |
| 16M | 3 | 29540 | 2.877 | 0.959 | 0.007 | 128M | 3 | 235672 | 2.876 | 0.959 | 0.006 |
| 16M | 4 | 22166 | 3.834 | 0.958 | 0.005 | 128M | 4 | 175724 | 3.857 | 0.964 | 0.002 |
| 16M | 5 | 17895 | 4.749 | 0.950 | 0.006 | 128M | 5 | 140989 | 4.807 | 0.961 | 0.002 |
| 16M | 6 | 15004 | 5.664 | 0.944 | 0.006 | 128M | 6 | 119072 | 5.692 | 0.949 | 0.004 |
| 16M | 7 | 12877 | 6.599 | 0.943 | 0.005 | 128M | 7 | 101317 | 6.689 | 0.956 | 0.002 |
| 16M | 8 | 11520 | 7.377 | 0.922 | 0.008 | 128M | 8 | 90134 | 7.519 | 0.940 | 0.004 |
| 32M | seq | 170452 | | | | 256M | seq | 1350158 | | | |
| 32M | 1 | 171926 | 0.991 | 0.991 | | 256M | 1 | 1364950 | 0.989 | 0.989 | |
| 32M | 2 | 87999 | 1.937 | 0.968 | 0.024 | 256M | 2 | 704288 | 1.917 | 0.959 | 0.032 |
| 32M | 3 | 58833 | 2.897 | 0.966 | 0.013 | 256M | 3 | 469923 | 2.873 | 0.958 | 0.016 |
| 32M | 4 | 44433 | 3.836 | 0.959 | 0.011 | 256M | 4 | 355856 | 3.794 | 0.949 | 0.014 |
| 32M | 5 | 35588 | 4.790 | 0.958 | 0.009 | 256M | 5 | 284293 | 4.749 | 0.950 | 0.010 |
| 32M | 6 | 29624 | 5.754 | 0.959 | 0.007 | 256M | 6 | 237631 | 5.682 | 0.947 | 0.009 |
| 32M | 7 | 25824 | 6.601 | 0.943 | 0.009 | 256M | 7 | 202138 | 6.679 | 0.954 | 0.006 |
| 32M | 8 | 22568 | 7.553 | 0.944 | 0.007 | 256M | 8 | 180758 | 7.469 | 0.934 | 0.008 |
| 64M | seq | 339088 | | | | 512M | seq | 2704213 | | | |
| 64M | 1 | 342194 | 0.991 | 0.991 | | 512M | 1 | 2805340 | 0.964 | 0.964 | |
| 64M | 2 | 176237 | 1.924 | 0.962 | 0.030 | 512M | 2 | 1402986 | 1.927 | 0.964 | 0.000 |
| 64M | 3 | 118196 | 2.869 | 0.956 | 0.018 | 512M | 3 | 944822 | 2.862 | 0.954 | 0.005 |
| 64M | 4 | 88214 | 3.844 | 0.961 | 0.010 | 512M | 4 | 709384 | 3.812 | 0.953 | 0.004 |
| 64M | 5 | 71601 | 4.736 | 0.947 | 0.012 | 512M | 5 | 565657 | 4.781 | 0.956 | 0.002 |
| 64M | 6 | 59347 | 5.714 | 0.952 | 0.008 | 512M | 6 | 475636 | 5.685 | 0.948 | 0.003 |
| 64M | 7 | 50980 | 6.651 | 0.950 | 0.007 | 512M | 7 | 405143 | 6.675 | 0.954 | 0.002 |
| 64M | 8 | 45032 | 7.530 | 0.941 | 0.008 | 512M | 8 | 357483 | 7.565 | 0.946 | 0.003 |

**Table 9.2** FindKeySeq/FindKeySmp3 running time-metrics without JIT compiler

| *N* | *K* | *T* | *Spdup* | *Eff* | *EDSF* |
|---:|---:|---:|---:|---:|---:|
| 16M | seq | 2634676 | | | |
| 16M | 1 | 2638909 | 0.998 | 0.998 | |
| 16M | 2 | 1322203 | 1.993 | 0.996 | 0.002 |
| 16M | 3 | 884351 | 2.979 | 0.993 | 0.003 |
| 16M | 4 | 664967 | 3.962 | 0.991 | 0.003 |
| 16M | 5 | 532116 | 4.951 | 0.990 | 0.002 |
| 16M | 6 | 443904 | 5.935 | 0.989 | 0.002 |
| 16M | 7 | 381755 | 6.901 | 0.986 | 0.002 |
| 16M | 8 | 338981 | 7.772 | 0.972 | 0.004 |

# CHAPTER 10

# Measuring Sizeup

in which we define further metrics for a parallel program’s performance, based on size rather than speed; we predict what the theoretical metrics should look like; we measure the actual metrics for our parallel program; and we compare two approaches for scaling up a parallel program

# CHAPTER 10 Measuring Sizeup

## 10.1 Sizeup Metrics

In Chapter 8, we expressed a program's running time as a function of the problem size and the number of processors: $T(N,K)$. However, we can just as easily turn that relationship around and express the program's problem size as a function of the running time and the number of processors: $N(T,K)$. Given a certain value for $T$, there is some problem size $N$ such that the program's running time on $K$ processors will be exactly $T$. We write $N_{\text{seq}}(T,K)$ for the sequential version's problem size and $N_{\text{par}}(T,K)$ for the parallel version's problem size.

We can now define a metric analogous to speedup, but one focusing on problem size instead of speed. A program's **sizeup** is the size of the parallel version running on $K$ processors relative to the size of the sequential version running on one processor for a given running time $T$:

$$Sizeup(T,K) = \frac{N_{\text{par}}(T,K)}{N_{\text{seq}}(T,1)} \tag{10.1}$$

Ideally, a parallel program should be able to solve twice as large a problem on two processors as on one processor, thrice as large a problem on three processors, four times as large a problem on four processors, and so on. That is, ideally, *Sizeup* should equal $K$. Because a plot of the ideal *Sizeup* versus $K$ is a straight line with unity slope, this ideal sizeup is also called a **linear sizeup**.

As with speedup, real parallel programs usually fall short of this ideal. **Sizeup efficiency** is a metric that captures how close to ideal a program's sizeup is:

$$SizeupEff(T,K) = \frac{Sizeup(T,K)}{K} \tag{10.2}$$

An ideal parallel program will have a sizeup efficiency of 1 for all running times $T$ and all numbers of processors $K$. A real parallel program typically has a sizeup efficiency less than 1, so that the sizeup is less than $K$—a **sublinear sizeup**.

## 10.2 Gustafson's Law

In his 1988 paper "Reevaluating Amdahl's Law" which we already mentioned, John Gustafson made two key observations based on Sandia National Laboratories' work with real-world parallel programs on large (1,024-processor) parallel computers. Quoting Gustafson:

- "One does not take a fixed-size problem and run it on various numbers of processors except when doing academic research; in practice, *the problem size scales with the number of processors.*"

- "As a first approximation, we have found that it is the *parallel* or *vector* part of a program that scales with the problem size."

In other words, the running time for the sequential portion of the program is the same no matter what the problem size. Based on these observations, Gustafson recommended that when running a parallel program on a computer with more processors, one should make the problem size larger to keep the running time the same. The picture looks like Figure 10.1, rather than the picture that led to Amdahl's Law.

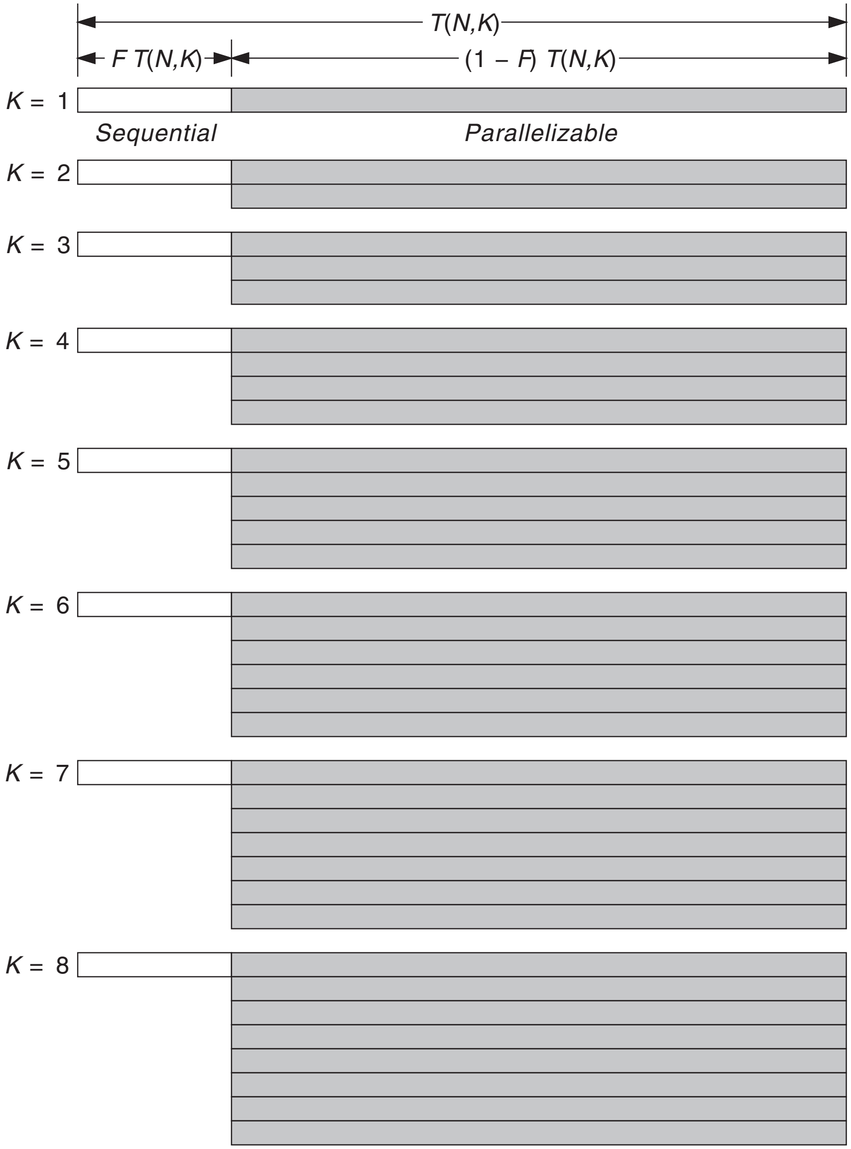


**Figure 10.1** A parallel program with running time $T(N,K)$ and sequential fraction $F$, where the problem size scales with the number of processors $K$

It's possible to derive a formula for the program's speedup when running on $K$ processors, based on how long the program *would have taken* to run on one processor with the larger problem size. If the program's running time on $K$ processors is $T(N,K)$ (kept the same regardless of what $K$ is), and if a fraction $F$ of that running time must be executed sequentially, then the running time on one processor for the same problem size would be the following:

$$T(N,1) = F \cdot T(N,K) + K \cdot (1-F) \cdot T(N,K) \tag{10.3}$$

Equation 10.3 is known as **Gustafson's Law**. The corresponding formulas for speedup and efficiency follow:

$$\begin{aligned} Speedup(N,K) &= \frac{T(N,1)}{T(N,K)} \\ &= \frac{F \cdot T(N,K) + K \cdot (1-F) \cdot T(N,F)}{T(N,K)} \\ &= F + K - KF \end{aligned} \tag{10.4}$$

$$Eff(N,K) = \frac{Speedup(N,K)}{K} = \frac{F}{K} + 1 - F \tag{10.5}$$

As the number of processors increases, Gustafson's Law predicts that the speedup continues increasing without limit, becoming approximately $K \cdot (1–F)$ as $K$ goes to infinity. Likewise, the efficiency becomes approximately $(1–F)$ as $K$ goes to infinity. This seemingly contradicts Amdahl's Law, which predicts that the speedup approaches a limit of $1/F$ and the efficiency approaches a limit of 0 as $K$ goes to infinity.

Gustafson's paper resulted in much confusion and controversy in the parallel computing community over whether Amdahl's Law really was "broken." In fact, there is no contradiction. The two laws are predicated on different assumptions. Amdahl's Law defines the sequential fraction $F$ with respect to the program's running time on *one* processor; Gustafson's Law defines the sequential fraction $F$ with respect to the program's running time on $K$ processors. Each law is valid under its own assumptions.

Actually, the speedup from Gustafson's Law is not all that meaningful. When scaling the problem size with the number of processors, we would never run the larger problem on a single processor in the first place, so there's not much point in considering the speedup with respect to a single processor. Our goal now is to achieve a *sizeup* rather than a speedup, so we are interested in a theoretical formula for *problem size* as a function of $K$, analogous to Amdahl's Law for running time as a function of $K$.

## 10.3 The Problem Size Laws

Suppose we assume, as Gustafson did, that a parallel program's sequential portion's running time is independent of $N$, that is, constant. Let's further assume that the program's parallelizable portion's running time (on one processor) is directly proportional to $N$. Then the program's **running time model** is given by this formula,

$$T(N,K) = a + \frac{1}{K} dN \tag{10.6}$$

where *N* is the problem size, *K* is the number of processors, and *a* and *d* are **model parameters** that are determined empirically for a particular program. (Later, we will see how to determine the model parameters.) Then, to get the program's **problem size model**, solve Equation 10.6 for *N*:

$$N(T,K) = \frac{1}{d}K(T-a) \tag{10.7}$$

Equation 10.7 is the **First Problem Size Law**. From it, we can derive formulas for sizeup and sizeup efficiency:

$$\begin{aligned} \mathit{Sizeup}(T,K) &= \frac{N(T,K)}{N(T,1)} \\ &= \frac{\frac{1}{d}K(T-a)}{\frac{1}{d}(T-a)} \\ &= K \end{aligned} \tag{10.8}$$

$$\mathit{SizeupEff}(T,K) = \frac{\mathit{Sizeup}(T,K)}{K} = 1 \tag{10.9}$$

Thus, the program should experience nothing but ideal sizeups when scaling to larger numbers of processors—if the assumptions behind the First Problem Size Law hold true, that the sequential portion's running time is constant and the parallelizable portion's running time is proportional to *N*.

However, this assumption, as Gustafson admits, is only an approximation. In reality, a parallel program's sequential portion's running time does increase somewhat as the problem size increases. Thus, a more realistic running-time model assumes that both the sequential portion's running time and the parallelizable portion's running time are general linear functions of *N*,

$$T(N,K) = (a+bN) + \frac{1}{K}(c+dN) \tag{10.10}$$

where (*a*+*bN*) is the running time for the sequential portion, (*c*+*dN*) is the running time for the parallelizable portion, and there are now four model parameters, *a*, *b*, *c*, and *d*. Solving Equation 10.10 for *N* gives the problem size model:

$$N(T,K) = \frac{T-a-\frac{c}{K}}{b+\frac{d}{K}} = \frac{KT-Ka-c}{Kb+d} \tag{10.11}$$

Equation 10.11 is the **Second Problem Size Law**. The corresponding formula for sizeup follows:

$$\begin{aligned} \mathit{Sizeup}(T,K) &= \frac{N(T,K)}{N(T,1)} \\ &= \frac{(KT - Ka - c)(b + d)}{(T - a - c)(Kb + d)} \end{aligned} \tag{10.12}$$

The constant portions of the running-time model, *a* and *c*, are usually negligible compared to the program's total running time *T*. With this assumption, the sizeup is the following:

$$\begin{aligned} \mathit{Sizeup}(T,K) &= \frac{(KT)(b + d)}{(T)(Kb + d)} \\ &= \frac{Kb + Kd}{Kb + d} \end{aligned} \tag{10.13}$$

Let's define the quantity *G* to be the ratio of the model parameters *b*/*d*, that is, the rate at which the sequential portion's running time grows with *N* relative to the rate at which the parallelizable portion's running time grows with *N*. Then, we can express sizeup and sizeup efficiency in terms of *G*:

$$\begin{aligned} \mathit{Sizeup}(\mathrm{T,K}) &= \frac{K\frac{b}{d} + K}{K\frac{b}{d} + 1} \\ &= \frac{KG + K}{KG + 1} \end{aligned} \tag{10.14}$$

$$\mathit{SizeupEff}(T,K) = \frac{\mathit{Sizeup}(T,K)}{K} = \frac{G + 1}{KG + 1} \tag{10.15}$$

Under the Second Problem Size Law, the sizeups are no longer ideal as *K* scales up. In the limit as *K* goes to infinity, the sizeup efficiency goes to 0, and the sizeup becomes the following:

$$\begin{aligned} \lim_{K\to\infty} \mathit{Sizeup}(T,K) &= \frac{KG + K}{KG} \\ &= 1 + \frac{1}{G} \end{aligned} \tag{10.16}$$

The sizeup approaches a fixed upper bound that depends on the reciprocal of *G*. Thus, 1/*G* plays the same role in limiting the sizeup under the Second Problem Size Law as 1/*F* played in limiting the speedup under Amdahl's Law. Plots of *Sizeup* and *SizeupEff* versus *K* for various values of *G* (Figures 10.2 and 10.3) closely resemble those for *Speedup* and *Eff* versus *K* (Figures 8.2 and 8.3).

Having derived the theoretical problem size formulas, we next look at how to measure a parallel program's actual sizeup and sizeup efficiency.

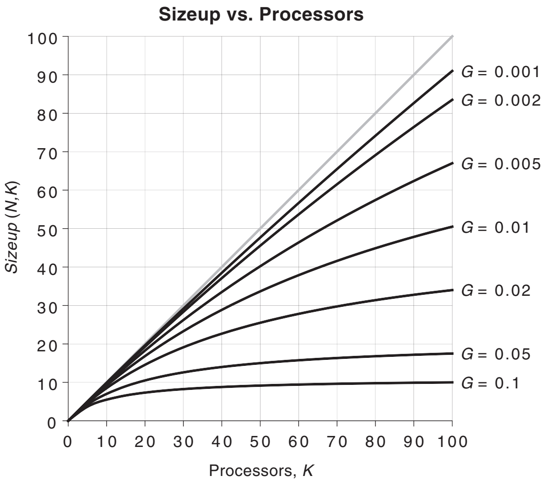


**Figure 10.2** Sizeup predicted by the Second Problem Problem Size Law

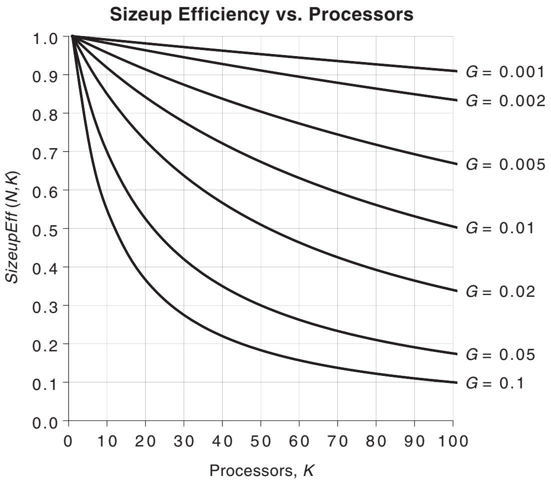


**Figure 10.3** Sizeup efficiency predicted by the Second Problem Size Law

## 10.4 Measuring Sizeup

While we can't directly measure the problem size that would make a program run for a certain amount of time, we can determine *N* as a function of *T* and *K* from our running-time measurements. Instead of plotting our data as *T* versus *K* for various values of *N*, suppose we plot our data as *N* versus *T* for various values of *K*. Table 10.1 (at the end of the chapter) tabulates, and Figure 10.4 plots, the FindKeySmp3 program's running-time data this way. Note that each curve is a straight line with a slope of 1. On a log-log plot, this means that *N* equals some constant times *T* to the power 1. This is what we expect. The amount of computation is defined to be proportional to the problem size *N* (the number of keys tested), therefore the running time *T* should be proportional to *N* and vice versa.

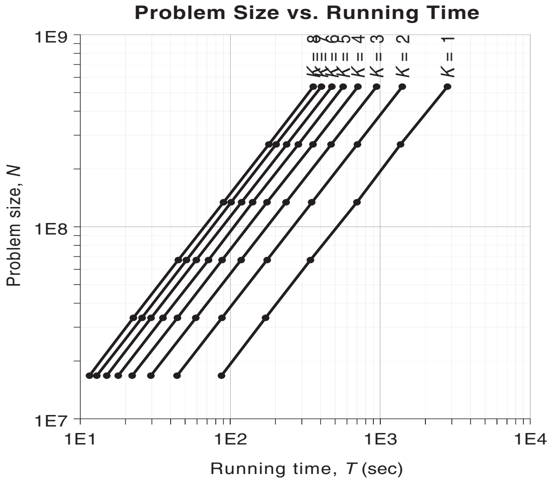


**Figure 10.4** FindKeySeq/FindKeySmp3 *N* versus *T*

We can now read problem size off the plot. Suppose we want to find $N$ for $T = 200$ seconds and $K = 1$ processor; that is, $N(200,1)$. We go to $T = 200$ on the horizontal axis, move up until we intersect the $K = 1$ curve, move left to the vertical axis, and find $N$ is about $3.9\times10^7$. If the program were to run for 200 seconds on one processor, it would test 39 million keys.

Instead of squinting at the plot, we can calculate $N$ by **interpolation** in the tabulated data. First, find two entries in the table, $(T_1, N_1)$ and $(T_2, N_2)$, that bracket the desired $T$ value—such that $T_1 \le T \le T_2$. For our example with $T = 200{,}000$ msec (200 seconds) and $K = 1$, the relevant table entries are (171926, 33554432) and (342194, 67108864). Then, calculate $N$ using the linear interpolation formula:

$$N = \frac{(T - T_1)}{(T_2 - T_1)}(N_2 - N_1) + N_1 \tag{10.17}$$

Substituting the values $T = 200000$, $T_1 = 171926$, $N_1 = 33554432$, $T_2 = 342194$, and $N_2 = 67108864$ into Equation 10.17, we compute $N = 39086929$.

Using interpolation, we can compute tables of $N$ versus $K$ for selected values of $T$, completely analogous to the tables of $T$ versus $K$ for selected values of $N$ that we compiled from our running-time measurements. From these tables, we can calculate sizeup and sizeup efficiency and gain insight into the program's performance as it scales up to larger problem sizes running on more processors.

However, when we select values of $T$ to analyze, we must make sure that the chosen values fall between the lowest and the highest $T$ values in the tables for *every* value of $K$. For Figure 10.4, this would be roughly from $T = 90$ seconds (the smallest $T$ value in the table for $K = 1$) to $T = 350$ seconds (the largest $T$ value in the table for $K = 8$). If we choose values for $T$ much outside this range, the linear interpolation formula will still work, but, in fact, we will be *extrapolating* beyond our measured data, not interpolating. Extrapolation is always more dangerous than interpolation.

## 10.5 FindKeySmp3 Sizeup Data

Table 10.2 (at the end of the chapter) gives the interpolated problem sizes for selected running times for the AES key search program (the one designed to avoid cache interference), as well as the sizeups and sizeup efficiencies. The chosen running times are $T = 80$, 100, 150, 200, 300, and 400 seconds. In the $K$ column, "seq" denotes the sequential version of the program (FindKeySeq) and 1 through 8 denote the parallel version of the program (FindKeySmp3).

Figure 10.5 plots the $N$ versus $K$ data from Table 10.2 for the AES key search program. Each curve is nearly a straight line with unity slope, indicating the ideal situation where the problem size that can be solved in a given amount of time is directly proportional to the number of processors. The plots of *Sizeup* versus $K$ and *SizeupEff* versus $K$ confirm that the program is achieving near-linear sizeups and sizeup efficiencies greater than 94 percent for all the selected running-time values.

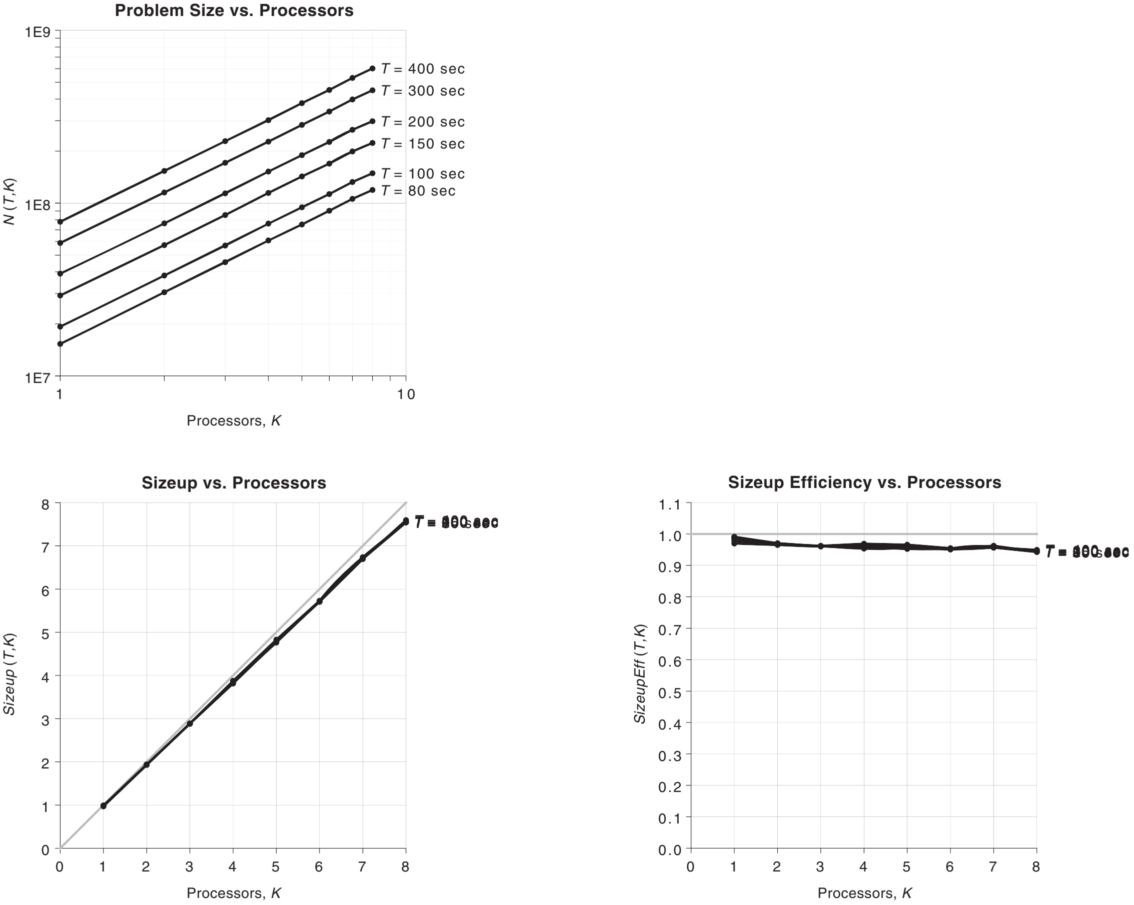


**Figure 10.5** FindKeySeq/FindKeySmp3 problem size metrics

We can also use the running-time data to calculate values for the model parameters $a$, $b$, $c$, and $d$ in the Second Problem Size Law's running-time model (Equation 10.10). From these, we can calculate the value $G = b/d$ that determines the maximum possible sizeup (Equation 10.16). The model parameters are calculated by doing a **general linear least squares curve fit** on the data. See Appendix C for a description of the general linear least squares algorithm. A Java program, class TimeFit, in the Parallel Java Library implements the algorithm. The fitted model parameters for the AES key search program are $a = 0$, $b = 2.26\times10^{-5}$, $c = 0$, and $d = 5.18\times10^{-3}$ (all times are in msec). The AES key search program's $G$ is $4.36\times10^{-3}$, and its maximum sizeup is 229. Refer to the Parallel Java documentation for instructions on how to use the TimeFit program.

By the way, the problem size metric plots in Figures 10.4 and 10.5 were produced by another Java program, class Sizeup, in the Parallel Java Library. The program takes the raw running-time measurements as in Table 9.1, calculates and prints the problem size, sizeup, and sizeup efficiency metrics, and generates the plots. Refer to the Parallel Java documentation for instructions on how to use the Sizeup program.

## 10.6 Speedup or Sizeup?

In Chapters 8 and 9 and in this chapter, we took a close look at the two approaches for utilizing a parallel computer as the number of processors increases:

- Reduce the running time while keeping the same problem size—go for *speedup.*
- Increase the problem size while keeping the same running time—go for *sizeup.*

We have Gustafson's testimony, with which most parallel computing practitioners agree, that on real-world problems one goes for sizeup, not speedup, as one scales up the number of processors. On the other hand, we have seen how Amdahl's Law predicts, and actual running time data verifies, that a parallel program's speedup diminishes as the number of processors increases, approaching an upper bound that depends on the program's sequential fraction *F*. Likewise, we have seen how the Second Problem Size Law predicts, and actual running time data verifies, that a parallel program's sizeup diminishes as the number of processors increases, approaching an upper bound that depends on the program's *G* value. Sizeup and speedup both diminish as the number of processors scales up, so why prefer sizeup to speedup?

One reason has nothing to do with the metrics per se, but rather with time constraints. Often, the problem size one *does* calculate is smaller than the problem size one would *like to* calculate, because the latter takes too long. For example, a parallel weather-modeling program used to generate tomorrow's forecast needs to finish well before tomorrow arrives, otherwise the forecast is useless. This time constraint in turn puts a limit on the problem size—the number of 3-D cells modeling the atmosphere, the number of time steps—that can be calculated, and therefore puts a limit on the accuracy of the solution. If we double the number of processors in the parallel computer, we don't particularly care that we can get the same forecast in half the time. We'd rather increase the problem size—cover the atmospheric region and time span with more, smaller cells and more, smaller time steps—to increase the forecast's accuracy.

Apart from such considerations, do the metrics themselves support the notion that going for sizeup is preferable to going for speedup? To gain some insight into this question, let's look more closely at the plots of *Eff* versus *K* and *SizeupEff* versus *K* for the AES key search program (Figures 10.6 and 10.7). We expanded the vertical scale to separate the curves for the various *T* and *N* values. Comparing the two plots, a slight but definite difference is apparent: As the number of processors scales up, the sizeups do not diminish quite as much as the speedups. This is true of most parallel programs. For programs other than the AES key search program—programs that have larger sequential fractions—the sizeups can be considerably larger than the speedups when adding more processors. This is another reason going for sizeup is preferable to going for speedup.

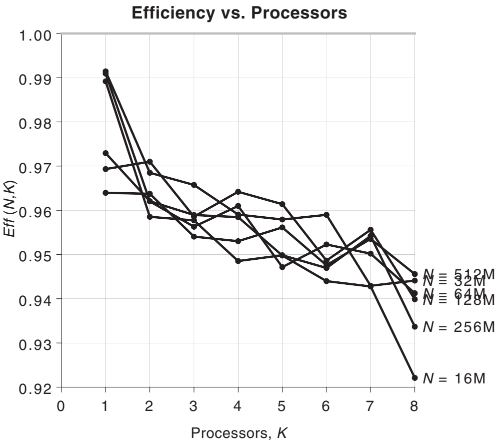


**Figure 10.6** *Eff* vs. K

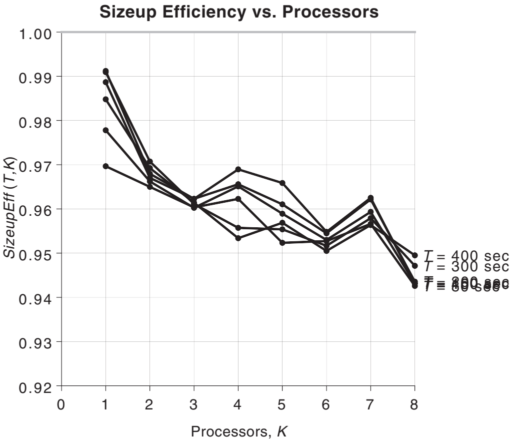


**Figure 10.7** *SizeupEff* vs. *K*

Why, then, worry about speedup at all? Why not focus solely on sizeup? Speedup is important during the parallel program's *development* stage. Precisely because speedup tends to be more sensitive to the number of processors and to the sequential fraction than sizeup tends to be, focusing on a program's speedup magnifies any design flaws that adversely affect the program's performance. Once these flaws are fixed and the program is yielding good speedups, the program will then experience good sizeups during the *operational* stage.

This book is concerned with designing parallel programs, so we will focus more on speedup as the performance metric. Just keep in mind that when solving actual problems on a parallel computer, you usually want to go for sizeup rather than speedup.

## 10.7 For Further Information

Amdahl's original paper on parallel program performance:

- G. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the AFIPS Spring Joint Computer Conference*, 1967, pages 483–485.

Gustafson's original paper on parallel program performance:

- J. Gustafson. Reevaluating Amdahl's law. *Communications of the ACM*, 31(5):532–533, May 1988.

A critique of Amdahl's and Gustafson's work:

- Y. Shi. Reevaluating Amdahl's law and Gustafson's law. October 1996. http://joda.cis.temple.edu/~shi/docs/amdahl/amdahl.html

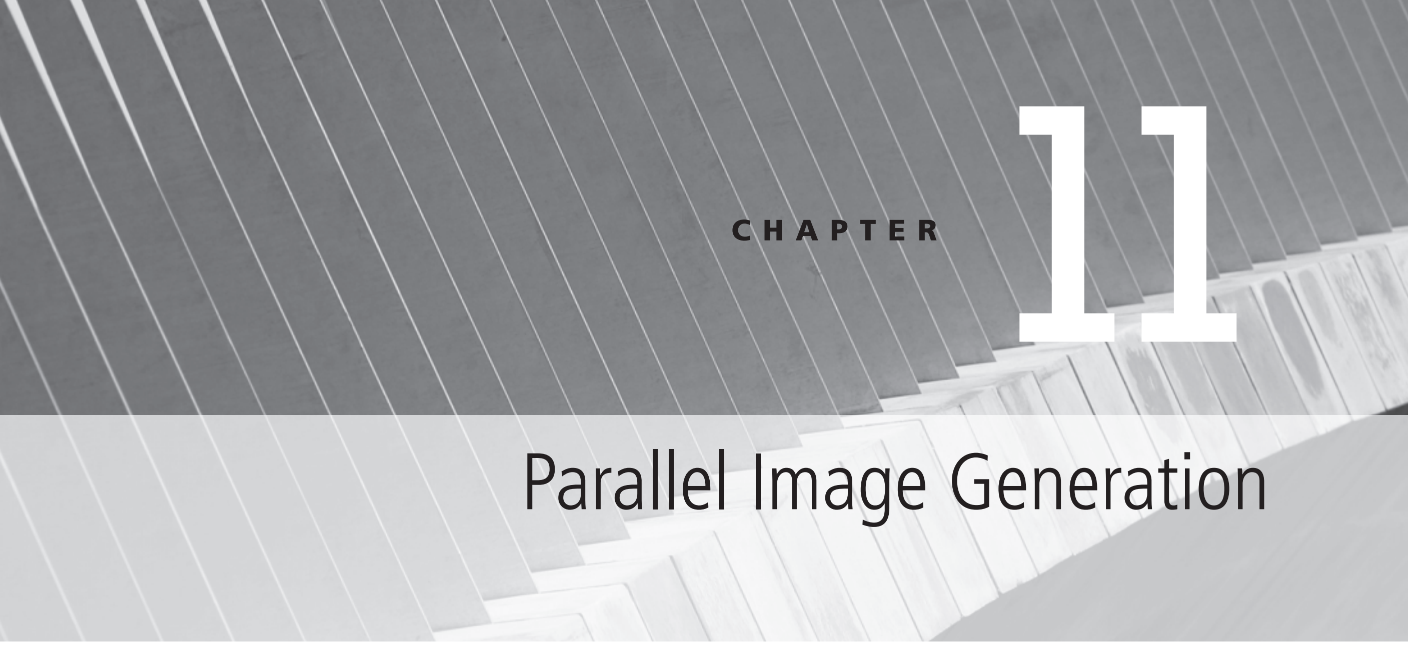
**Table 10.1** FindKeySeq/FindKeySmp3 running-time metrics, rearranged

| *K* | *T* | *N* | *K* | *T* | *N* |
|---:|---:|---:|---:|---:|---:|
| seq | 84981 | 16777216 | | | |
| seq | 170452 | 33554432 | | | |
| seq | 339088 | 67108864 | | | |
| seq | 677725 | 134217728 | | | |
| seq | 1350158 | 268435456 | | | |
| seq | 2704213 | 536870912 | | | |
| 1 | 87346 | 16777216 | 5 | 17895 | 16777216 |
| 1 | 171926 | 33554432 | 5 | 35588 | 33554432 |
| 1 | 342194 | 67108864 | 5 | 71601 | 67108864 |
| 1 | 699172 | 134217728 | 5 | 140989 | 134217728 |
| 1 | 1364950 | 268435456 | 5 | 284293 | 268435456 |
| 1 | 2805340 | 536870912 | 5 | 565657 | 536870912 |
| 2 | 44166 | 16777216 | 6 | 15004 | 16777216 |
| 2 | 87999 | 33554432 | 6 | 29624 | 33554432 |
| 2 | 176237 | 67108864 | 6 | 59347 | 67108864 |
| 2 | 348984 | 134217728 | 6 | 119072 | 134217728 |
| 2 | 704288 | 268435456 | 6 | 237631 | 268435456 |
| 2 | 1402986 | 536870912 | 6 | 475636 | 536870912 |
| 3 | 29540 | 16777216 | 7 | 12877 | 16777216 |
| 3 | 58833 | 33554432 | 7 | 25824 | 33554432 |
| 3 | 118196 | 67108864 | 7 | 50980 | 67108864 |
| 3 | 235672 | 134217728 | 7 | 101317 | 134217728 |
| 3 | 469923 | 268435456 | 7 | 202138 | 268435456 |
| 3 | 944822 | 536870912 | 7 | 405143 | 536870912 |
| 4 | 22166 | 16777216 | 8 | 11520 | 16777216 |
| 4 | 44433 | 33554432 | 8 | 22568 | 33554432 |
| 4 | 88214 | 67108864 | 8 | 45032 | 67108864 |
| 4 | 175724 | 134217728 | 8 | 90134 | 134217728 |
| 4 | 355856 | 268435456 | 8 | 180758 | 268435456 |
| 4 | 709384 | 536870912 | 8 | 357483 | 536870912 |

**Table 10.2** FindKeySeq/FindKeySmp3 problem size metrics

| *T* | *K* | *N* | *Sizeup* | *SizeEff* | *T* | *K* | *N* | *Sizeup* | *SizeEff* |
|---:|---:|---:|---:|---:|---:|---:|---:|---:|---:|
| 80000 | seq | 15799489 | | | 200000 | seq | 39433760 | | |
| 80000 | 1 | 15320070 | 0.970 | 0.970 | 200000 | 1 | 39086929 | 0.991 | 0.991 |
| 80000 | 2 | 30492790 | 1.930 | 0.965 | 200000 | 2 | 76340329 | 1.936 | 0.968 |
| 80000 | 3 | 45518899 | 2.881 | 0.960 | 200000 | 3 | 113839886 | 2.887 | 0.962 |
| 80000 | 4 | 60813528 | 3.849 | 0.962 | 200000 | 4 | 152305961 | 3.862 | 0.966 |
| 80000 | 5 | 75231988 | 4.762 | 0.952 | 200000 | 5 | 189487102 | 4.805 | 0.961 |
| 80000 | 6 | 90315216 | 5.716 | 0.953 | 200000 | 6 | 225834326 | 5.727 | 0.954 |
| 80000 | 7 | 105798083 | 6.696 | 0.957 | 200000 | 7 | 265589248 | 6.735 | 0.962 |
| 80000 | 8 | 119138990 | 7.541 | 0.943 | 200000 | 8 | 297662985 | 7.548 | 0.944 |
| 100000 | seq | 19725315 | | | 300000 | seq | 59331310 | | |
| 100000 | 1 | 19287253 | 0.978 | 0.978 | 300000 | 1 | 58793762 | 0.991 | 0.991 |
| 100000 | 2 | 38118075 | 1.932 | 0.966 | 300000 | 2 | 115188393 | 1.941 | 0.971 |
| 100000 | 3 | 56823729 | 2.881 | 0.960 | 300000 | 3 | 171075449 | 2.883 | 0.961 |
| 100000 | 4 | 76147203 | 3.860 | 0.965 | 300000 | 4 | 226816724 | 3.823 | 0.956 |
| 100000 | 5 | 94575063 | 4.795 | 0.959 | 300000 | 5 | 283420727 | 4.777 | 0.955 |
| 100000 | 6 | 112787837 | 5.718 | 0.953 | 300000 | 6 | 338778730 | 5.710 | 0.952 |
| 100000 | 7 | 132461915 | 6.715 | 0.959 | 300000 | 7 | 397839316 | 6.705 | 0.958 |
| 100000 | 8 | 148829664 | 7.545 | 0.943 | 300000 | 8 | 449557429 | 7.577 | 0.947 |
| 150000 | seq | 29539882 | | | 400000 | seq | 79180005 | | |
| 150000 | 1 | 29205210 | 0.989 | 0.989 | 400000 | 1 | 77975906 | 0.985 | 0.985 |
| 150000 | 2 | 57131670 | 1.934 | 0.967 | 400000 | 2 | 153489258 | 1.938 | 0.969 |
| 150000 | 3 | 85277088 | 2.887 | 0.962 | 400000 | 3 | 228371993 | 2.884 | 0.961 |
| 150000 | 4 | 114490743 | 3.876 | 0.969 | 400000 | 4 | 301954201 | 3.814 | 0.953 |
| 150000 | 5 | 142657380 | 4.829 | 0.966 | 400000 | 5 | 378825774 | 4.784 | 0.957 |
| 150000 | 6 | 169230556 | 5.729 | 0.955 | 400000 | 6 | 451564367 | 5.703 | 0.951 |
| 150000 | 7 | 199026862 | 6.738 | 0.963 | 400000 | 7 | 530070274 | 6.694 | 0.956 |
| 150000 | 8 | 222881642 | 7.545 | 0.943 | 400000 | 8 | 601451873 | 7.596 | 0.950 |

*This page intentionally left blank*



CHAPTER 11

# Parallel Image Generation

in which we encounter the issues and trade-offs that must be addressed when a parallel program generates an image and writes it to a file; we study an example of a program that generates large images; and we observe this program’s curious and unsatisfactory performance

# CHAPTER 11 Parallel Image Generation

## 11.1 The Mandelbrot Set

For our second programming problem, we will explore generating an image in parallel. Rather than render a frame of a computer-animated film, we'll generate an image that's much easier to compute: the **Mandelbrot Set** (Figure 11.1). Published by IBM mathematician Benoit Mandelbrot in his 1977 book *The Fractal Geometry of Nature*, the Mandelbrot Set is perhaps the most famous fractal object ever discovered.

The Mandelbrot Set is a set of points in the plane, defined as follows. Given a point ($x,y$), compute a sequence of other points ($a_i$,$b_i$), $i$=0, 1, 2, ... using the following formulas:

$$\begin{aligned} a_0 &= 0 \\ b_0 &= 0 \\ a_{i+1} &= a_i^2 - b_i^2 + x \\ b_{i+1} &= 2a_i b_i + y \end{aligned} \tag{11.1}$$

If each point in the infinite sequence ($a_i$,$b_i$) stays finite, then ($x,y$) is a member of the Mandelbrot Set. If the sequence of points ($a_i$,$b_i$) shoots off to infinity, then ($x,y$) is not a member of the Mandelbrot Set. In Figure 11.1, points in the Mandelbrot Set are black, and points not in the Mandelbrot Set are shades of gray. An alternative image renders points not in the Mandelbrot set with a range of colors.

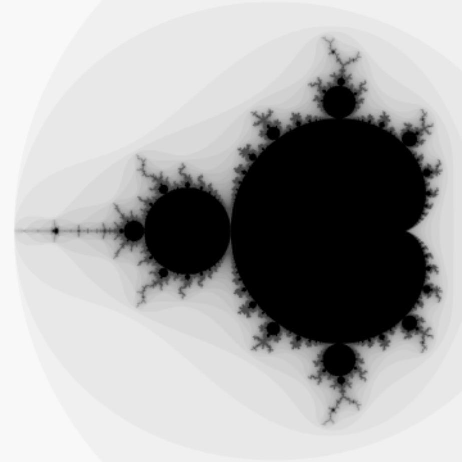


**Figure 11.1** The Mandelbrot Set

A computer program for computing an image of the Mandelbrot Set needs a different criterion for deciding whether a point is in the set; it is not possible to compute an infinite sequence of points and still get the answer in a finite time. It can be proven that if the point $(a_i,b_i)$ ever exceeds a distance of 2 from the origin—if $(a_i^2 + b_i^2)^{1/2} > 2$ for some $i$—then the sequence will inevitably shoot off to infinity. So set a limit on the number of points, say 1,000 points, and start computing the sequence. If $(a_i,b_i)$ exceeds a distance of 2 from the origin before $i$ reaches the limit on the number of points, then $(x,y)$ is not a member of the Mandelbrot Set. If $i$ reaches the limit before $(a_i,b_i)$ exceeds a distance of 2, then $(x,y)$ is assumed to be a member of the Mandelbrot Set. Further iterations may reveal that $(x,y)$ is, in fact, not in the set, but the program has to stop somewhere.

The color of a point not in the Mandelbrot Set depends on the value of $i$ when the iteration stops. The color's hue is given by the formula $(i/i_{\max})^{\gamma}$, where $i_{\max}$ is the iteration limit and $\gamma$ is a parameter that adjusts the range of hues displayed. In the grayscale version, the gray shade of a point not in the Mandelbrot Set likewise depends on $(i/i_{\max})$.

## 11.2 Color Images

The program we are about to study, our second parallel program, computes an image of the Mandelbrot Set and stores it in a file for later viewing. But, before we can design the program, we need to take a detour and consider how a program can create color images.

Each pixel in the image can have a different color, so we need a way to set the color of each individual pixel. However, the graphics classes in the standard Java platform are not well suited to this task. Class Graphics2D in package java.awt is primarily intended for displaying graphical user interfaces and is oriented toward drawing lines, rectangles, ellipses, text, and so on, not pixels. While the classes in package java.awt.image do you let you manipulate pixels, these classes are so general and flexible that it takes multiple objects and several layers of method calls just to set the color of one pixel. Performance is paramount in parallel programs, and a program that uses package java.awt.image to manipulate pixels would be too slow.

The Parallel Java Library takes a different approach. Creating a color image is done in two steps. The first step is to create an integer matrix (type `int[][]`) containing the pixel data. The rows and columns of elements in the matrix correspond one-for-one with the rows and columns of pixels in the image. Each element in the matrix stores the corresponding pixel's color in packed **red-green-blue (RGB) format** (Figure 11.2).

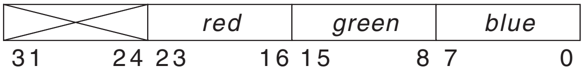


**Figure 11.2** Packed RGB format

The pixel's color is specified by three components, red, green, and blue. Each component is an 8-bit unsigned integer in the range 0 through 255, with 0 being fully dark and 255 being fully bright. The components are packed into an `int` with the red component in bits 23 through 16, green in bits 15 through 8, and blue in bits 7 through 0. Bits 31 through 24 are unused and are set to zero. Class edu.rit.color.IntRGB has a static `pack()` method that packs the three separate components together into an `int`.

A color can also be specified as **hue-saturation-brightness (HSB)** components instead of RGB. The hue component is a real number between 0 and 1, where a hue of 0 gives red, 1/6 gives yellow, 2/6 gives green, 3/6 gives cyan, 4/6 gives blue, 5/6 gives magenta, 1 gives red again, and other values produce intermediate colors. The saturation component is a real number between 0 and 1 that specifies how gray or colored the color is. A saturation of 0 yields fully gray; a saturation of 1 yields fully colored; intermediate saturation values yield mixtures of gray and colored. The brightness component is a real number between 0 and 1 that specifies how dark or light the color is. A brightness of 0 yields fully dark (black); a brightness of 1 yields fully light (somewhere between white and colored depending on the saturation); intermediate brightness values yield somewhere between a gray shade and a darkened color (depending on the saturation). Class edu.rit.color.HSB has a static `pack()` method that converts the three HSB components to RGB components and packs the RGB components together into an `int`.

After allocating an integer matrix for the pixel data and setting the color of each pixel by storing the desired packed RGB value into each matrix element, the second step is to store the image in a file. One way is simply to write the integers in the matrix to the file. With an integer occupying four bytes, a 1,000×1,000-pixel image file requires 4,000,000 bytes. However, this would be an inefficient way to store the pixel data. In a typical image, the data is redundant; there are many regions where all the pixels have the same value (color). Such data can be **compressed** to remove the redundancy and reduce the file size.

For this reason, widely used image file formats like the Joint Photographic Expert Group's JPEG format and the Portable Network Graphics (PNG) format store the pixel data in compressed form. JPEG uses **lossy compression**; when the image is reconstructed from the compressed data, the pixel values turn out to be slightly different from their original values. While this is fine for photographs where the human eye can't tell the difference, lossy compression is not appropriate for non-photographic images like the Mandelbrot Set image. PNG, on the other hand, uses **lossless compression**; the reconstructed pixels are identical to the originals.

However, there's a problem with using PNG to generate images in a parallel program. Any data-compression scheme involves a trade-off between short running time and small compressed file size. It takes a longer-running algorithm to produce a smaller file. PNG was designed for *network* graphics, where an image file is generated once, stored on a Web server, and downloaded millions of times. In that setting, it's sensible to spend time making the image file size as small as possible. The reduction in the time required to download millions of copies of the image file more than compensates for the time required to run the compression algorithm. Thus, PNG uses sophisticated compression techniques that take a long time to compress large images. But, in a parallel program, this is the wrong trade-off. Because the pixel data must be compressed and written to the image file sequentially, using PNG increases the parallel program's sequential fraction. Depending on the time required to compress and write the pixel data relative to the time required to calculate the pixel data, the sequential fraction could be large and the resulting performance reduction severe.

In a parallel program, we need a different trade-off: an image-compression technique that runs faster than PNG, at the price of increased image file size. We will use the **Parallel Java Graphics (PJG)** format. On the Mandelbrot Set images, PJG file sizes are 30–50 percent larger than the corresponding PNG files. However, to compress and write the pixel data, the PJG algorithm takes only one-tenth to one-twentieth as much time as the PNG algorithm. The Parallel Java Library provides class edu.rit.image.PJGImage and class edu.rit.image.PJGColorImage for creating color images, writing them to PJG files, and reading them from PJG files. For further information about the PJG image file format and compression algorithm, refer to the Parallel Java documentation.

To store an image in a PJG file, construct an instance of class edu.rit.image.PJGColorImage, specifying the image height, image width, and pixel data matrix as constructor arguments.

```
int[][] pixeldata = new int [ROWS] [COLS];
PJGColorImage image = new PJGColorImage (ROWS, COLS, pixeldata);
```

Then, call the image's `prepareToWrite()` method, specifying the output stream on which to write the PJG file.

```
PJGImage.Writer writer = image.prepareToWrite (outputstream);
```

The `prepareToWrite()` method returns a writer object, an instance of class PJGImage.Writer. Call the writer's `write()` method to write the pixel data, and then close the writer.

```
writer.write();
writer.close();
```

The Parallel Java Library includes a program to display an image stored in a PJG file. To run the program, type this command, specifying the PJG filename:

```
$ java PJG file.pjg
```

The program has menu items to convert the image to a PNG file or a PostScript file, should you desire to use those formats.

Package edu.rit.image also has class PJGGrayImage for creating **grayscale images**, where each pixel is one of 256 shades of gray. Class PJGGrayImage uses a byte array (type `byte[][]`) to store the pixel data, which takes one-fourth as much storage as an integer array of the same height and width. Likewise, grayscale image files tend to be smaller than full-color image files for an image of the same height and width. For further information about class PJGGrayImage, refer to the Parallel Java documentation.

## 11.3 Sequential Program

Class MandelbrotSetSeq in package edu.rit.smp.fractal is a sequential program for calculating an image of the Mandelbrot Set and storing the image in a PJG file. The program can calculate different regions of the Mandelbrot Set at different resolutions by specifying these command-line arguments:

- *width*—Image width in pixels.
- *height*—Image height in pixels.
- *xcenter*—X coordinate of the image's center point.
- *ycenter*—Y coordinate of the image's center point.
- *resolution*—Image resolution in pixels per unit. The distance between adjacent pixels is 1/*resolution*.

- *maxiter*—Maximum number of iterations for deciding whether a point is in the set ($i_{max}$).
- *gamma*—Exponent in the formula for calculating pixel hues ($\gamma$).
- *filename*—Output PJG image filename.

A color image corresponding to Figure 11.1 is produced with this command:

```
$ java edu.rit.smp.fractal.MandelbrotSetSeq 400 400 -0.75 0 150 \
  1000 0.4 fig11_1.pjg
```

Figure 11.3 is a magnified view (at higher resolution) of one of the little blobs off the main blob. A color image corresponding to Figure 11.3 is produced with this command:

```
$ java edu.rit.smp.fractal.MandelbrotSetSeq 400 400 -0.55 0.6 9600 \
  1000 0.6 fig11_3.pjg
```

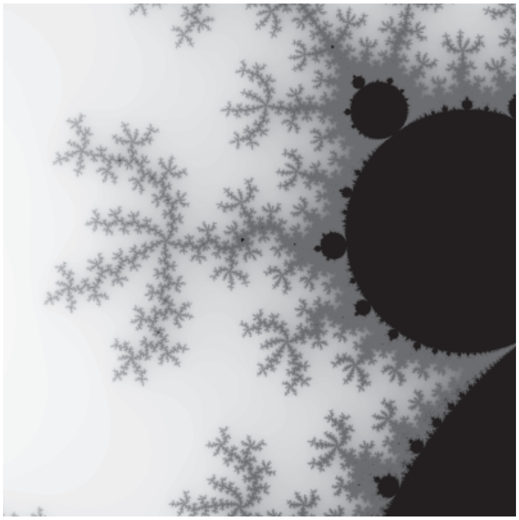


**Figure 11.3** A piece of the Mandelbrot Set

Here is the source code for class MandelbrotSetSeq.

```
package edu.rit.smp.fractal;
import edu.rit.color.HSB;
import edu.rit.image.PJGColorImage;
import edu.rit.image.PJGImage;
import java.io.BufferedOutputStream;
import java.io.File;
import java.io.FileOutputStream;
public class MandelbrotSetSeq
   {
```

```
   // Command line arguments.
   static int width;
   static int height;
   static double xcenter;
   static double ycenter;
   static double resolution;
   static int maxiter;
   static double gamma;
   static File filename;

   // Initial pixel offsets from center.
   static int xoffset;
   static int yoffset;

   // Image matrix.
   static int[][] matrix;
   static PJGColorImage image;

   // Table of hues.
   static int[] huetable;

   /**
    * Mandelbrot Set main program.
    */
   public static void main
      (String[] args)
      throws Exception
      {
      // Start timing.
      long t1 = System.currentTimeMillis();

      // Validate command line arguments.
      if (args.length != 8) usage();
      width = Integer.parseInt (args[0]);
      height = Integer.parseInt (args[1]);
      xcenter = Double.parseDouble (args[2]);
      ycenter = Double.parseDouble (args[3]);
      resolution = Double.parseDouble (args[4]);
      maxiter = Integer.parseInt (args[5]);
      gamma = Double.parseDouble (args[6]);
      filename = new File (args[7]);
```

The variables `xoffset` and `yoffset` store the horizontal and vertical distances, in pixels, from the top-left pixel to the center of the image.

```
// Initial pixel offsets from center.
xoffset = -(width - 1) / 2;
yoffset = (height - 1) / 2;
```

Here are the two steps for creating a color image: allocate an integer matrix with one element for each pixel, and create an instance of class PJGColorImage.

```
// Create image matrix to store results.
matrix = new int [height] [width];
image = new PJGColorImage (height, width, matrix);
```

After determining the number of iterations *i* for each pixel inside the loop, we could compute the pixel's RGB value using the formula $(i/i_{\text{max}})^{\gamma}$ also inside the loop. However, it will save time in the loop if we pre-calculate the RGB value corresponding to each *i* value just once outside the loop and store the RGB value in a table indexed by *i*. Then, inside the loop, we can simply look up the proper RGB value in the table, which is faster than calculating the RGB value. This **table-lookup** technique often can be used to reduce a computation's running time, at the price of more storage to hold the table. For this program, the table lookup yields about a 10 percent savings in the calculation's running time.

```
// Create table of hues for different iteration counts.
huetable = new int [maxiter+1];
for (int i = 0; i < maxiter; ++ i)
   {
   huetable[i] = HSB.pack
      (/*hue*/ (float) Math.pow
         (((double) i) / ((double) maxiter), gamma),
       /*sat*/ 1.0f,
       /*bri*/ 1.0f);
   }
huetable[maxiter] = HSB.pack (1.0f, 1.0f, 0.0f);
```

The program measures the running time for the three parts of the program—pre-calculation, calculation, and post-calculation—as well as the total running time. Here is the first of the three time snapshots, as we begin the pixel calculations.

```
      long t2 = System.currentTimeMillis();

      // Compute all rows and columns.
      for (int r = 0; r < height; ++ r)
         {
         int[] matrix_r = matrix[r];
```

In a Java program, a matrix is stored as an array of rows, where each row is a reference to an array of successive column elements (Figure 11.4). When accessing a matrix row by row, setting up a reference to the current row can reduce the program's running time. In this program, `matrix_r` is a reference to row `r` of the pixel data matrix. Accessing the matrix element in row `r`, column `c` with the expression `matrix_r[c]` requires only one array index operation; the expression `matrix[r][c]` requires two array index operations.

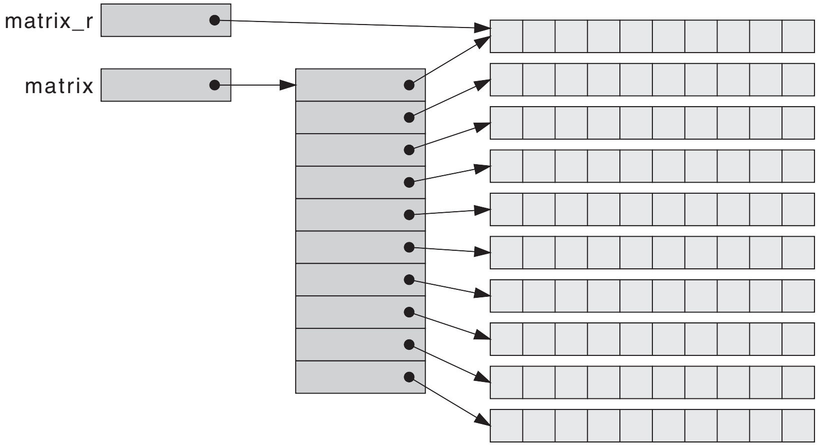


**Figure 11.4** Storage for pixel data matrix and reference to one row

```
         double y = ycenter + (yoffset - r) / resolution;

         for (int c = 0; c < width; ++ c)
            {
            double x = xcenter + (xoffset + c) / resolution;

            // Iterate until convergence.
            int i = 0;
            double aold = 0.0;
            double bold = 0.0;
            double a = 0.0;
            double b = 0.0;
            double zmagsqr = 0.0;
```

Note that we don't have to store all the points ($a_i$,$b_i$) for every value of *i*; we need to store only the previous point (`aold`, `bold`) and the current point (`a`,`b`). The variable `zmagsqr` stores the square of the distance of the point ($a_i$,$b_i$) from the origin, namely $a_i^2 + b_i^2$. We compute the square of the distance rather than the distance itself, $(a_i^2 + b_i^2)^{1/2}$, to eliminate all the time taken by the square root calculations. Because the square root function is monotonic, comparing the squared distance to $2^2$ is equivalent to comparing the distance to 2. By the way, it is *much, much* faster to compute $a^2$ with the expression `a*a` than with the expression `Math.pow(a,2)`.

```
      while (i < maxiter && zmagsqr <= 4.0)
         {
         ++ i;
         a = aold*aold - bold*bold + x;
         b = 2.0*aold*bold + y;
         zmagsqr = a*a + b*b;
         aold = a;
         bold = b;
         }
```

Here is where we do the table lookup to get the RGB value for the current pixel, and use the row reference `matrix_r` to store it.

```
      // Record number of iterations for pixel.
      matrix_r[c] = huetable[i];
      }
   }
long t3 = System.currentTimeMillis();
```

Here we use the PJG color image object to write the PJG image file. Using a BufferedOutputStream on top of the FileOutputStream causes the program to write the file in large chunks of bytes. Using a FileOutputStream directly would cause the program to write the file one byte at a time, which is slower than writing it in large chunks.

```
// Write image to PJG file.
PJGImage.Writer writer =
   image.prepareToWrite
      (new BufferedOutputStream
         (new FileOutputStream (filename)));
writer.write();
writer.close();

// Stop timing.
long t4 = System.currentTimeMillis();
System.out.println ((t2-t1) + " msec pre");
```

```
      System.out.println ((t3-t2) + " msec calc");
      System.out.println ((t4-t3) + " msec post");
      System.out.println ((t4-t1) + " msec total");
      }
   }
```

## 11.4 Parallel Program

Calculating the Mandelbrot Set is a result parallel problem where the results are the pixel colors. Like the AES key search problem, calculating the Mandelbrot Set is another example of a massively parallel problem. The calculations for each pixel do not depend in any way on the calculations for any other pixel. We can calculate all the pixels in parallel, in any order we please.

To parallelize the computation, we divide the image rows among the parallel team threads. Each thread will compute a different subset of the rows and will compute all the columns within each row. The outer loop will become a parallel for loop and the inner loop will stay as it is. The code before and after the outer loop will be done in a single thread as in the sequential version. (Because calculating the hue table takes little time, it's not worth the effort to parallelize it.)

The variables in the parallel version are treated as follows:

- `width`, `height`, `xcenter`, `ycenter`, `resolution`, `maxiter`, `gamma`, `filename`, `xoffset`, and `yoffset`—The command-line arguments will be program shared variables. They are WORM variables, so they need no synchronization.
- `matrix`—The pixel data matrix is the principal shared variable in the program. It will be a program shared variable because all the threads as well as the main program must access it. Each matrix element is written by just one thread, so there is no need to synchronize the threads when they access the matrix. Because the threads are no longer active when the main program is reading the matrix elements to write the PJG image file, there is no need to synchronize the threads with the main program. There's no need to add padding bytes to the pixel data matrix, because it is a shared variable. (We add padding to *per-thread* variables to eliminate *false* sharing.)
- `image`—It is not strictly necessary for the image object to be a program shared variable because the threads never access it. However, for readability, we will declare it immediately after the pixel data matrix.
- `huetable`—This also will be a program shared variable because the main program writes it and all the threads read it. It is a WORM variable and needs no synchronization.
- The remaining variables are all thread local variables or main program local variables that need no synchronization.

Class MandelbrotSetSmp in package edu.rit.smp.fractal is the SMP parallel version of the program.

```
package edu.rit.smp.fractal;
import edu.rit.color.HSB;
import edu.rit.image.PJGColorImage;
import edu.rit.image.PJGImage;
import edu.rit.pj.IntegerForLoop;
import edu.rit.pj.ParallelRegion;
import edu.rit.pj.ParallelTeam;
import java.io.BufferedOutputStream;
import java.io.File;
import java.io.FileOutputStream;
public class MandelbrotSetSmp
   {
   // Command line arguments.
   static int width;
   static int height;
   static double xcenter;
   static double ycenter;
   static double resolution;
   static int maxiter;
   static double gamma;
   static File filename;

   // Initial pixel offsets from center.
   static int xoffset;
   static int yoffset;

   // Image matrix.
   static int[][] matrix;
   static PJGColorImage image;

   // Table of hues.
   static int[] huetable;

   /**
    * Mandelbrot Set main program.
    */
   public static void main
      (String[] args)
      throws Exception
      {
      // Start timing.
      long t1 = System.currentTimeMillis();
```

```
        // Validate command line arguments.
        if (args.length != 8) usage();
        width = Integer.parseInt (args[0]);
        height = Integer.parseInt (args[1]);
        xcenter = Double.parseDouble (args[2]);
        ycenter = Double.parseDouble (args[3]);
        resolution = Double.parseDouble (args[4]);
        maxiter = Integer.parseInt (args[5]);
        gamma = Double.parseDouble (args[6]);
        filename = new File (args[7]);

        // Initial pixel offsets from center.
        xoffset = -(width - 1) / 2;
        yoffset = (height - 1) / 2;

        // Create image matrix to store results.
        matrix = new int [height] [width];
        image = new PJGColorImage (height, width, matrix);

        // Create table of hues for different iteration counts.
        huetable = new int [maxiter+1];
        for (int i = 0; i < maxiter; ++ i)
           {
           huetable[i] = HSB.pack
              (/*hue*/ (float) Math.pow
                 (((double) i) / ((double) maxiter), gamma),
               /*sat*/ 1.0f,
               /*bri*/ 1.0f);
           }
        huetable[maxiter] = HSB.pack (1.0f, 1.0f, 0.0f);

        long t2 = System.currentTimeMillis();
```

The outer loop is now a parallel for loop, inside a parallel region, executed by a parallel thread team.

```
        // Compute all rows and columns.
        new ParallelTeam().execute (new ParallelRegion()
           {
           public void run() throws Exception
              {
              execute (0, height-1, new IntegerForLoop()
                 {
                 public void run (int first, int last)
```

```
            {
            for (int r = first; r <= last; ++ r)
               {
               int[] matrix_r = matrix[r];
               double y = ycenter+(yoffset-r)/resolution;

               for (int c = 0; c < width; ++ c)
                  {
                  double x = xcenter+(xoffset+c)/resolution;

                  // Iterate until convergence.
                  int i = 0;
                  double aold = 0.0;
                  double bold = 0.0;
                  double a = 0.0;
                  double b = 0.0;
                  double zmagsqr = 0.0;
                  while (i < maxiter && zmagsqr <= 4.0)
                     {
                     ++ i;
                     a = aold*aold - bold*bold + x;
                     b = 2.0*aold*bold + y;
                     zmagsqr = a*a + b*b;
                     aold = a;
                     bold = b;
                     }

                  // Record number of iterations for pixel.
                  matrix_r[c] = huetable[i];
                  }
               }
            }
         });
      }
   });

long t3 = System.currentTimeMillis();

// Write image to PJG file.
PJGImage.Writer writer =
   image.prepareToWrite
      (new BufferedOutputStream
         (new FileOutputStream (filename)));
writer.write();
writer.close();
```

```
      // Stop timing.
      long t4 = System.currentTimeMillis();
      System.out.println ((t2-t1) + " msec pre");
      System.out.println ((t3-t2) + " msec calc");
      System.out.println ((t4-t3) + " msec post");
      System.out.println ((t4-t1) + " msec total");
      }
   }
```

Table 11.1 (at the end of the chapter) lists, and Figure 11.5 plots, the MandelbrotSetSmp program's running-time data. Each program run calculated the same area as Figure 11.1, with increasing image dimensions $n \times n$ pixels, and with resolutions $r$ increasing in proportion to $n$. Each run's problem size was the total number of pixels, $N = n^2$. The particular $n$, $r$, and $N$ values follow:

| $n$ | $r$ | $N$ |
|---:|---:|---:|
| 3200 | 1200 | 10M |
| 4480 | 1680 | 20M |
| 6400 | 2400 | 40M |
| 8960 | 3360 | 80M |
| 12800 | 4800 | 160M |
| 17920 | 6720 | 320M |

The other command-line arguments were *xcenter* = –0.75, *ycenter* = 0, *maxiter* = 1000, and *gamma* = 0.4.

The running time plot shows that something strange is going on in this program. The pattern is the same no matter what the problem size. The running time for two processors is a bit more than half the running time for one processor, more or less as expected. But then the running time for three processors is *larger* than the running time for two processors! Going to four processors reduces the running time again, although it is not half as much as for two processors. Going to five processors increases the running time, and so on. There's no point in worrying about the Mandelbrot Set program's speedup or efficiency until we figure out what is causing this strange behavior, and then fix it. That will be the topic of Chapter 12.

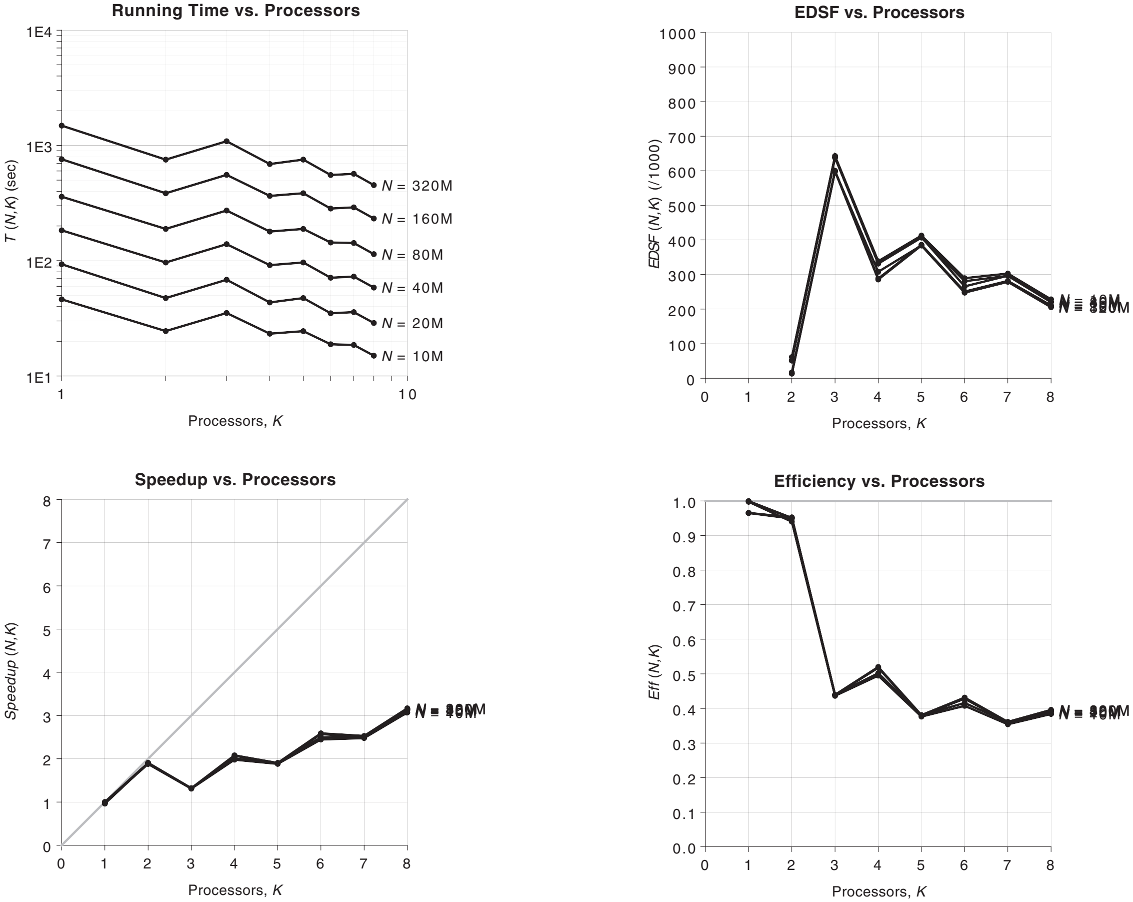


**Figure 11.5** MandelbrotSetSeq/MandelbrotSetSmp running time metrics

## 11.5 For Further Information

On the Mandelbrot Set and many other aspects of fractals:

- B. Mandelbrot. *The Fractal Geometry of Nature*. W. H. Freeman and Company, 1977.

**Table 11.1** MandelbrotSetSeq/MandelbrotSetSmp running-time metrics

| *N* | *K* | *T* | *Spdup* | *Eff* | *EDSF* | *N* | *K* | *T* | *Spdup* | *Eff* | *EDSF* |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 10M | seq | 46082 | | | | 80M | seq | 358869 | | | |
| 10M | 1 | 46198 | 0.997 | 0.997 | | 80M | 1 | 359057 | 0.999 | 0.999 | |
| 10M | 2 | 24514 | 1.880 | 0.940 | 0.061 | 80M | 2 | 188679 | 1.902 | 0.951 | 0.051 |
| 10M | 3 | 35195 | 1.309 | 0.436 | 0.643 | 80M | 3 | 272468 | 1.317 | 0.439 | 0.638 |
| 10M | 4 | 23277 | 1.980 | 0.495 | 0.338 | 80M | 4 | 179037 | 2.004 | 0.501 | 0.332 |
| 10M | 5 | 24488 | 1.882 | 0.376 | 0.413 | 80M | 5 | 188658 | 1.902 | 0.380 | 0.407 |
| 10M | 6 | 18840 | 2.446 | 0.408 | 0.289 | 80M | 6 | 143824 | 2.495 | 0.416 | 0.281 |
| 10M | 7 | 18589 | 2.479 | 0.354 | 0.303 | 80M | 7 | 142449 | 2.519 | 0.360 | 0.296 |
| 10M | 8 | 15002 | 3.072 | 0.384 | 0.228 | 80M | 8 | 114025 | 3.147 | 0.393 | 0.220 |
| 20M | seq | 89953 | | | | 160M | seq | 732332 | | | |
| 20M | 1 | 93200 | 0.965 | 0.965 | | 160M | 1 | 758750 | 0.965 | 0.965 | |
| 20M | 2 | 47396 | 1.898 | 0.949 | 0.017 | 160M | 2 | 384454 | 1.905 | 0.952 | 0.013 |
| 20M | 3 | 68348 | 1.316 | 0.439 | 0.600 | 160M | 3 | 555532 | 1.318 | 0.439 | 0.598 |
| 20M | 4 | 43444 | 2.071 | 0.518 | 0.288 | 160M | 4 | 364862 | 2.007 | 0.502 | 0.308 |
| 20M | 5 | 47417 | 1.897 | 0.379 | 0.386 | 160M | 5 | 384832 | 1.903 | 0.381 | 0.384 |
| 20M | 6 | 35004 | 2.570 | 0.428 | 0.251 | 160M | 6 | 283484 | 2.583 | 0.431 | 0.248 |
| 20M | 7 | 35804 | 2.512 | 0.359 | 0.282 | 160M | 7 | 290204 | 2.524 | 0.361 | 0.280 |
| 20M | 8 | 28828 | 3.120 | 0.390 | 0.211 | 160M | 8 | 232241 | 3.153 | 0.394 | 0.207 |
| 40M | seq | 183295 | | | | 320M | seq | 1433954 | | | |
| 40M | 1 | 183428 | 0.999 | 0.999 | | 320M | 1 | 1485352 | 0.965 | 0.965 | |
| 40M | 2 | 96494 | 1.900 | 0.950 | 0.052 | 320M | 2 | 752769 | 1.905 | 0.952 | 0.014 |
| 40M | 3 | 139167 | 1.317 | 0.439 | 0.638 | 320M | 3 | 1087533 | 1.319 | 0.440 | 0.598 |
| 40M | 4 | 91535 | 2.002 | 0.501 | 0.332 | 320M | 4 | 689470 | 2.080 | 0.520 | 0.286 |
| 40M | 5 | 96547 | 1.899 | 0.380 | 0.408 | 320M | 5 | 753350 | 1.903 | 0.381 | 0.384 |
| 40M | 6 | 71180 | 2.575 | 0.429 | 0.266 | 320M | 6 | 553974 | 2.588 | 0.431 | 0.248 |
| 40M | 7 | 72818 | 2.517 | 0.360 | 0.296 | 320M | 7 | 567188 | 2.528 | 0.361 | 0.279 |
| 40M | 8 | 58451 | 3.136 | 0.392 | 0.221 | 320M | 8 | 452692 | 3.168 | 0.396 | 0.205 |

*This page intentionally left blank*

# CHAPTER 12

# Load Balancing

in which we discover why it is important to give each thread in a parallel program an equal amount of work; we see that dividing the computations evenly among the threads does not always divide the work evenly; and we learn the programming constructs for splitting up the work equally

# CHAPTER 12 Load Balancing

## 12.1 Load Balance

To understand the strange running-time behavior of the Mandelbrot Set program from Chapter 11, we must first collect running-time data for each thread in the parallel team separately. Class MandelbrotSetSmp2 is a modified version of class MandelbrotSetSmp that collects this extra data. The program takes a snapshot of the system clock at the beginning, the end of the initialization section, the end of the pixel-calculation section, and the end of the PJG file-writing section. In addition, the program records the starting and ending times of the parallel region's `run()` method executed by each parallel team thread. The program also records the starting and ending times of the parallel for loop's `run()` method as each thread executes a certain chunk of loop indexes.

Here's what the program printed when it was run on one processor of the "parasite" computer to calculate a 3,200×3,200-pixel image.

```
$ java -Dpj.nt=1 edu.rit.smp.fractal.MandelbrotSetSmp2 3200 3200 \
  -0.75 0 1200 1000 0.4 ms3200.pjg
242 msec pre finish
        268 msec thread 0 start
                279 msec chunk 0 start
                45531 msec chunk 0 finish
        45535 msec thread 0 finish
45535 msec calc finish
46333 msec post finish
46333 msec total
```

Here's what the program printed when run on two processors. Because the pre-calculation section and post-calculation section run in a single thread, their running times are nearly the same as the one-processor case. The running time for the parallel calculation section decreased by a factor of about 2, as expected. To be precise, the speedup for the calculation section was 1.926, an efficiency of 0.963.

```
$ java -Dpj.nt=2 edu.rit.smp.fractal.MandelbrotSetSmp2 3200 3200 \
  -0.75 0 1200 1000 0.4 ms3200.pjg
242 msec pre finish
        267 msec thread 0 start
                283 msec chunk 0 start
                23752 msec chunk 0 finish
        23752 msec thread 0 finish
        267 msec thread 1 start
                283 msec chunk 0 start
                23705 msec chunk 0 finish
        23708 msec thread 1 finish
23753 msec calc finish
24528 msec post finish
24528 msec total
```

Here's what the program printed when run on three processors. Unlike the two-processor case where each thread's running time was about the same, here thread 1's running time is much larger than thread 0's or thread 2's running time.

```
$ java -Dpj.nt=3 edu.rit.smp.fractal.MandelbrotSetSmp2 3200 3200 \
  -0.75 0 1200 1000 0.4 ms3200.pjg
240 msec pre finish
        269 msec thread 0 start
                281 msec chunk 0 start
                5847 msec chunk 0 finish
        5847 msec thread 0 finish
        269 msec thread 1 start
                281 msec chunk 0 start
                34435 msec chunk 0 finish
        34435 msec thread 1 finish
        269 msec thread 2 start
                281 msec chunk 0 start
                5803 msec chunk 0 finish
        5806 msec thread 2 finish
34435 msec calc finish
35095 msec post finish
35095 msec total
```

Figure 12.1 depicts the program's running time for $K = 1$ through 8 processors. The pre-calculation time is shown as a white bar. The parallel team threads' calculation times are shown as dark gray bars. The post-calculation time is shown as a light gray bar.

**Load balance** is the extent to which each processor (thread) in a parallel program does the same amount of work. If each thread's running time is the same, as for $K = 2$ in Figure 12.1, the program is said to have a **balanced load**. If each thread's running time is not the same, as for $K = 3$ through 8, the program is said to have an **unbalanced load**.

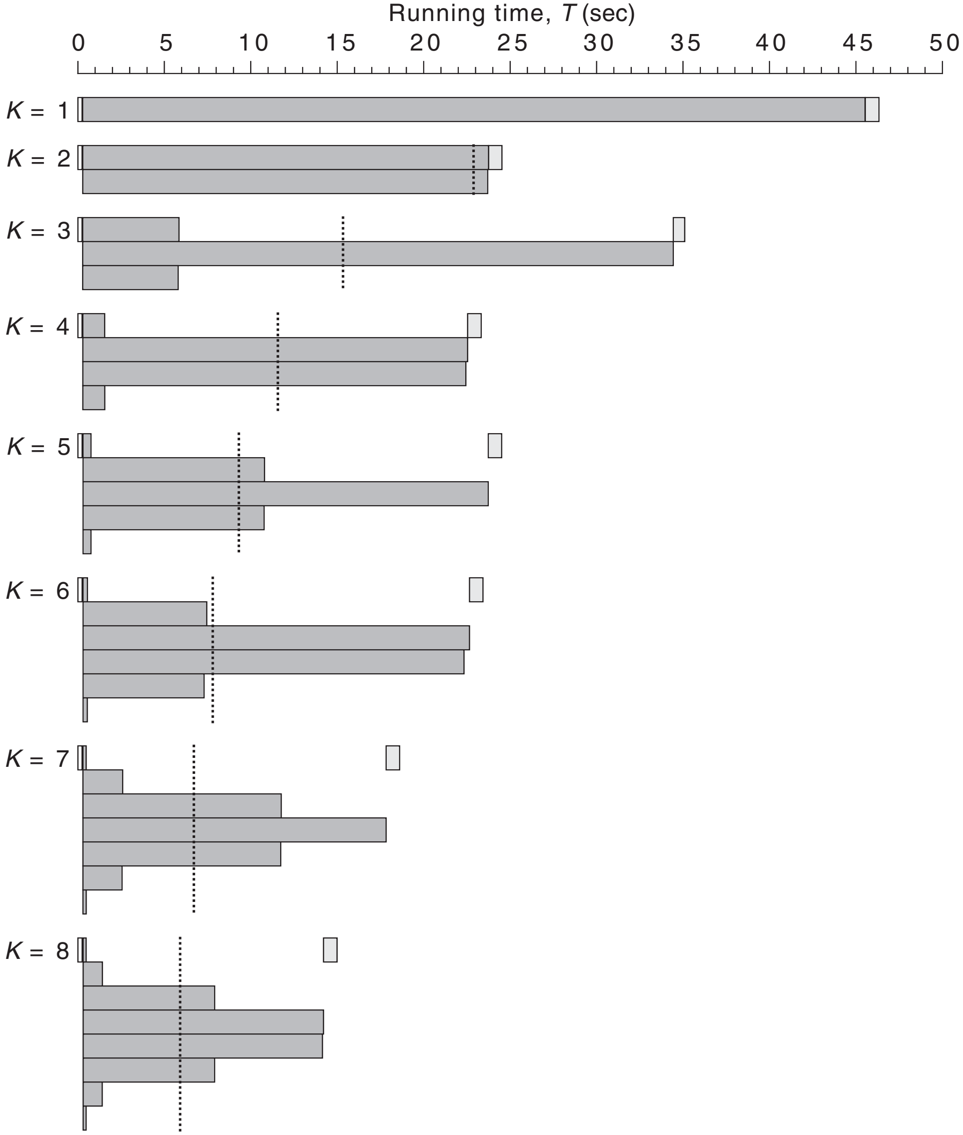


**Figure 12.1** Running times for each portion of the MandelbrotSetSmp program on different numbers of processors

To quantify the load balance, we divide the actual running time for the program's parallel portion by what the parallel portion running time *should have been* if the load were perfectly balanced:

$$B = \frac{T_p(K)}{T_p(1)/K} = \frac{KT_p(K)}{T_p(1)} \tag{12.1}$$

where $T_{\mathrm{p}}(K)$ is the running time for the program's parallel portion (not the whole program) on $K$ processors. If $B$ is 1, the load is perfectly balanced. The larger $B$ is, the more unbalanced the load. Here are the load balance values calculated from the Mandelbrot Set program's running time measurements.

| $K$ | $T_{\mathrm{p}}(K)$ | $T_{\mathrm{p}}(1)$ | $B$ |
|---|---|---|---|
| 2 | 23511 | 45293 | 1.04 |
| 3 | 34195 | 45293 | 2.26 |
| 4 | 22298 | 45293 | 1.97 |
| 5 | 23484 | 45293 | 2.59 |
| 6 | 22400 | 45293 | 2.97 |
| 7 | 17585 | 45293 | 2.72 |
| 8 | 13953 | 45293 | 2.46 |

In Figure 12.1, the dark gray bars depict the actual running time for the parallel portion, $T_{\mathrm{p}}(K)$, and the dotted lines depict what the running time should have been, $T_{\mathrm{p}}(1)/K$. This illustrates the effect of an unbalanced load on a parallel program's performance. If the load is not balanced, the program takes more time than it should when running on multiple processors, thus reducing the speedup and efficiency. *A balanced load is crucial for good parallel program performance.*

## 12.2 Achieving a Balanced Load

The AES key search program divided the computations equally among the threads and achieved good speedups and efficiencies, indicative of a balanced load. The Mandelbrot Set program also divides the computations equally among the threads, but does not achieve a balanced load. How can this be?

The answer lies in the nature of the calculations each program performs within its parallel loop. In the AES key search program, each loop iteration does exactly the same calculations—generate a full key, set the trial key into the cipher object, encrypt the plaintext, and compare the trial ciphertext to the correct ciphertext—and these calculations take the same amount of time in every loop iteration. Therefore, the AES key search program's load is inherently balanced.

In the Mandelbrot Set program, however, the calculations for each pixel do *not* take the same amount of time. Look back at Figure 11.1. The gray areas represent pixels that the program concluded were not in the Mandelbrot Set. Furthermore, for most of these pixels the program was able to reach its decision after just a few iterations. On the other hand, the black areas represent pixels that the program concluded were in the Mandelbrot Set; but the program could not reach this conclusion until it had done 1,000 iterations for each of these pixels. Thus, the program required much more running time to calculate the black pixels.

Now look at what happens when we run the program on a parallel computer (Figure 12.2). The parallel for loop divides the pixel rows evenly among the threads. For $K = 2$, each thread does about the same amount of work because of the image's symmetry. This results in a balanced load and good performance. But for $K = 3$, thread 1 gets the lion's share of the black pixels while threads 0 and 2 get mostly gray pixels. Because the black pixels take much longer to calculate, thread 1's running time ends up much longer than thread 0's and thread 2's. For larger numbers of processors, the middle threads continue to get more of the black pixels, resulting in an unbalanced load.

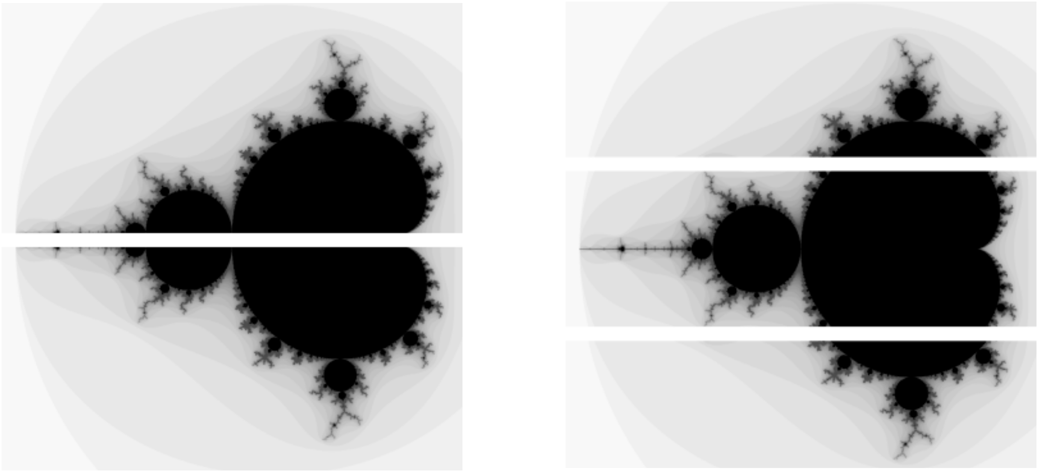


**Figure 12.2** Pixel rows divided among 2 and 3 threads

To achieve a balanced load for the Mandelbrot Set program, *the pixel rows cannot be divided evenly among the threads.* Some of the threads must calculate more of the rows, namely the rows with mostly gray pixels. The other threads must calculate fewer rows, namely the rows with many black pixels. In this way, all the threads end up computing for about the same amount of time.

## 12.3 Parallel For Loop Schedules

Parallel Java has the ability to divide the iterations of a parallel for loop *unevenly* among the parallel team threads, so as to achieve a balanced load when some iterations require more time than others. You do this by specifying the parallel for loop's **schedule**. There are three kinds of schedules: fixed, dynamic, and guided.

With a **fixed schedule**, Parallel Java divides the set of *N* loop iterations into *K* **chunks**, one chunk for each thread (Figure 12.3).

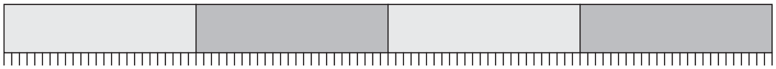


**Figure 12.3** Fixed schedule, 100 iterations, 4 threads

The **chunk size**, the number of iterations in the chunk, is the same for each chunk, namely *N/K* (plus or minus one in case *N* is not divisible by *K*). Parallel Java hands one chunk to each thread, and each thread calls the parallel for loop's `run()` method exactly once with the lower and upper bounds for the thread's own chunk passed in as arguments. This is called a "fixed" schedule because the set of iterations each thread will perform is predetermined, or fixed, before the iterations start.

With a **dynamic schedule**, Parallel Java divides the set of loop iterations into chunks of a specified size (Figure 12.4).

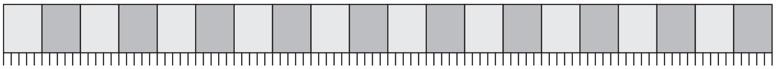


**Figure 12.4** Dynamic schedule, 100 iterations, chunk size 5

The default chunk size is 1, but you can specify a chunk size greater than 1. Parallel Java hands one chunk to each thread, which calls the `run()` method for that chunk. When the `run()` method returns, the thread gets the next available chunk and proceeds to execute it. The threads continue in this way until all the chunks have been executed. This is called a "dynamic" schedule because the set of iterations each thread performs is determined dynamically as the parallel loop progresses.

In the Mandelbrot Set program, the amount of computation in each loop iteration (each image row) depends on the region of the Mandelbrot Set being calculated. Different images result in different distributions of workload among the loop iterations. However, the dynamic schedule does not need to know the workload distribution ahead of time; the dynamic schedule adapts automatically to the workload distribution as the loop executes. If one thread is taking a long time to compute one chunk of iterations, the other threads are able to work on other chunks. With some threads executing a few long-running-time chunks and other threads executing many short-running-time chunks, it's more likely that all the threads will finish at about the same time, resulting in a more balanced load than a fixed schedule.

With a **guided schedule**, Parallel Java divides the set of loop iterations into chunks that start out large and get progressively smaller (Figure 12.5).

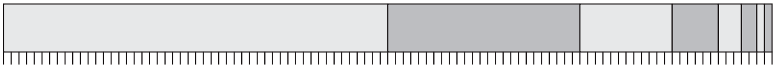


**Figure 12.5** Guided schedule, 100 iterations, 1 thread

Specifically, the size of each chunk is half the remaining number of iterations divided by the number of threads, except each chunk is no smaller than a specified minimum chunk size. The default minimum chunk size is 1, but you can specify a minimum chunk size greater than 1. For example, with $N = 100$ iterations, $K = 1$ thread, and a minimum chunk size of 1, the first chunk's size will be 100/2/1 = 50; the second chunk's size will be (100–50)/2/1 = 25; and successive chunk sizes will be 12, 6, 3, 2, 1, 1.

With a guided schedule, the more threads there are, the more chunks there are, and the smaller the chunks are. For example, with $N = 100$ iterations, $K = 2$ threads, and a minimum chunk size of 1, the first chunk's size will be 100/2/2 = 25; the second chunk's size will be (100–25)/2/2 = 18; and successive chunk sizes will be 14, 10, 8, 6, 4, 3, 3, 2, and seven chunks of size 1 (Figure 12.6).

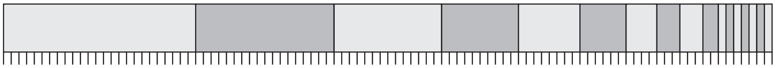


**Figure 12.6** Guided schedule, 100 iterations, 2 threads

Like a dynamic schedule, a guided schedule adapts automatically to the workload distribution as the loop executes, resulting in a more-balanced load than a fixed schedule.

Which parallel loop schedule should you use? Keep in mind that the goal is to achieve a balanced load while minimizing the loop overhead; the more chunks into which the loop iterations are partitioned, the more the overhead. If every loop iteration's computations will take the same amount of time, use a fixed schedule; this achieves a balanced load with as few chunks as possible (one chunk per thread). If different iterations will take different amounts of time, use a guided or dynamic schedule to balance the load. It's difficult to give more specific guidelines because parallel programs are too diverse. A guided schedule might give better performance on one program; a dynamic schedule might give better performance on another program. Increasing the chunk size might give better performance due to fewer chunks and less overhead, or might give worse performance due to increased load imbalance. You probably will have to try different schedules or chunk sizes to get the best performance for a particular program.

To specify the schedule for a parallel for loop, declare the parallel for loop subclass's `schedule()` method to return an object representing the desired schedule.

```
new IntegerForLoop()
   {
   public IntegerSchedule schedule()
      {
      return /* desired schedule object */;
      }
   . . .
   }
```

The possible schedule objects are the following:

- Fixed schedule—
  `return IntegerSchedule.fixed();`
- Dynamic schedule, default chunk size (1)—
  `return IntegerSchedule.dynamic();`
- Dynamic schedule, chunk size of, for example, 10—
  `return IntegerSchedule.dynamic(10);`
- Guided schedule, default minimum chunk size (1)—
  `return IntegerSchedule.guided();`
- Guided schedule, minimum chunk size of, for example, 10—
  `return IntegerSchedule.guided(10);`
- Custom schedule—
  `return new` *`classname`*`();`
  where *classname* is the name of the custom schedule subclass
- Runtime schedule (the default; discussed shortly)—
  `return IntegerSchedule.runtime();`

You can define your own subclass of class IntegerSchedule to implement a custom parallel for loop scheduling algorithm if the built-in ones don't do what you need. For further information, see the Parallel Java documentation for class IntegerSchedule.

If you do not declare the parallel for loop's `schedule()` method, the default schedule is a **runtime schedule**. You can also specify a runtime schedule explicitly. A runtime schedule is not a fourth kind of schedule; rather, it lets you specify the schedule at run time. A parallel for loop with a runtime schedule looks at the `"pj.schedule"` Java system property to determine the actual schedule. You can specify the schedule by including the `"-Dpj.schedule"` flag on the Java command line. For example:

```
$ java -Dpj.schedule=guided . . .
```

The possible values for the `"-Dpj.schedule"` flag are the following:

- Fixed schedule—
  `-Dpj.schedule=fixed`
- Dynamic schedule, default chunk size—
  `-Dpj.schedule=dynamic`
- Dynamic schedule, chunk size of 10—
  `-Dpj.schedule="dynamic(10)"`
- Guided schedule, default minimum chunk size—
  `-Dpj.schedule=guided`
- Guided schedule, minimum chunk size of 10—
  `-Dpj.schedule="guided(10)"`
- Custom schedule—
  `-Dpj.schedule=`*`classname`*
  where *classname* is the fully qualified class name of the custom schedule subclass. For further information, refer to the Parallel Java documentation.

Note that quotation marks might be needed. If a parallel for loop has a runtime schedule and the `"pj.schedule"` Java system property is not defined, the default is a fixed schedule. (This is why every parallel for loop we've studied up until this point has divided the iterations evenly among the threads.)

The default runtime schedule lets you experiment with different parallel for loop schedules from the Java command line without needing to recompile your program. Once you have decided on the best schedule, you can enshrine it in your program by declaring the `schedule()` method.

## 12.4 Parallel Program with Load Balancing

To achieve a balanced load in the Mandelbrot Set program, all that's needed is to switch the parallel for loop from the default fixed schedule to, let's say, a guided schedule. Table 12.1 (at the end of the chapter) lists, and Figure 12.7 plots, the MandelbrotSetSmp program's running time metrics for the same inputs as Figure 11.5, except the command line included the following flag:

```
$ java -Dpj.schedule=guided ...
```

The new data looks a lot better than the old data. No longer do the running times bounce up and down as *K* increases.

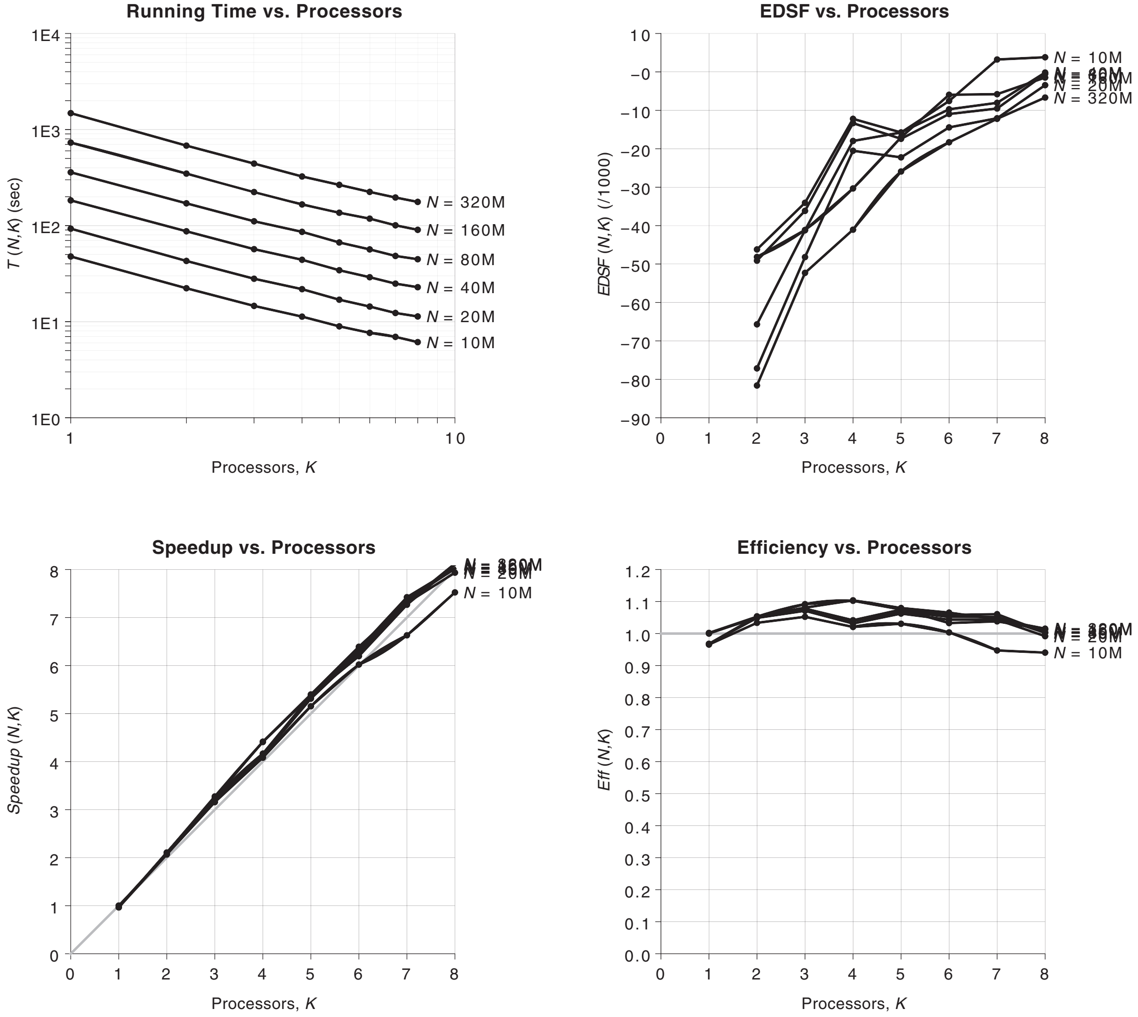


**Figure 12.7** MandelbrotSetSeq/MandelbrotSetSmp running-time metrics, guided schedule

To confirm that switching to a guided schedule has solved the load balance problem, Figure 12.8 depicts the program's running time for $K = 1$ through 8 processors with a guided schedule for a 3,200×3,200-pixel image. Each thread's calculation time is subdivided to show how long it took to calculate each chunk. The figure demonstrates why dividing the image into chunks using a guided schedule achieves a balanced load; with chunks of various running times distributed among the threads, all the threads finish processing chunks at about the same time.

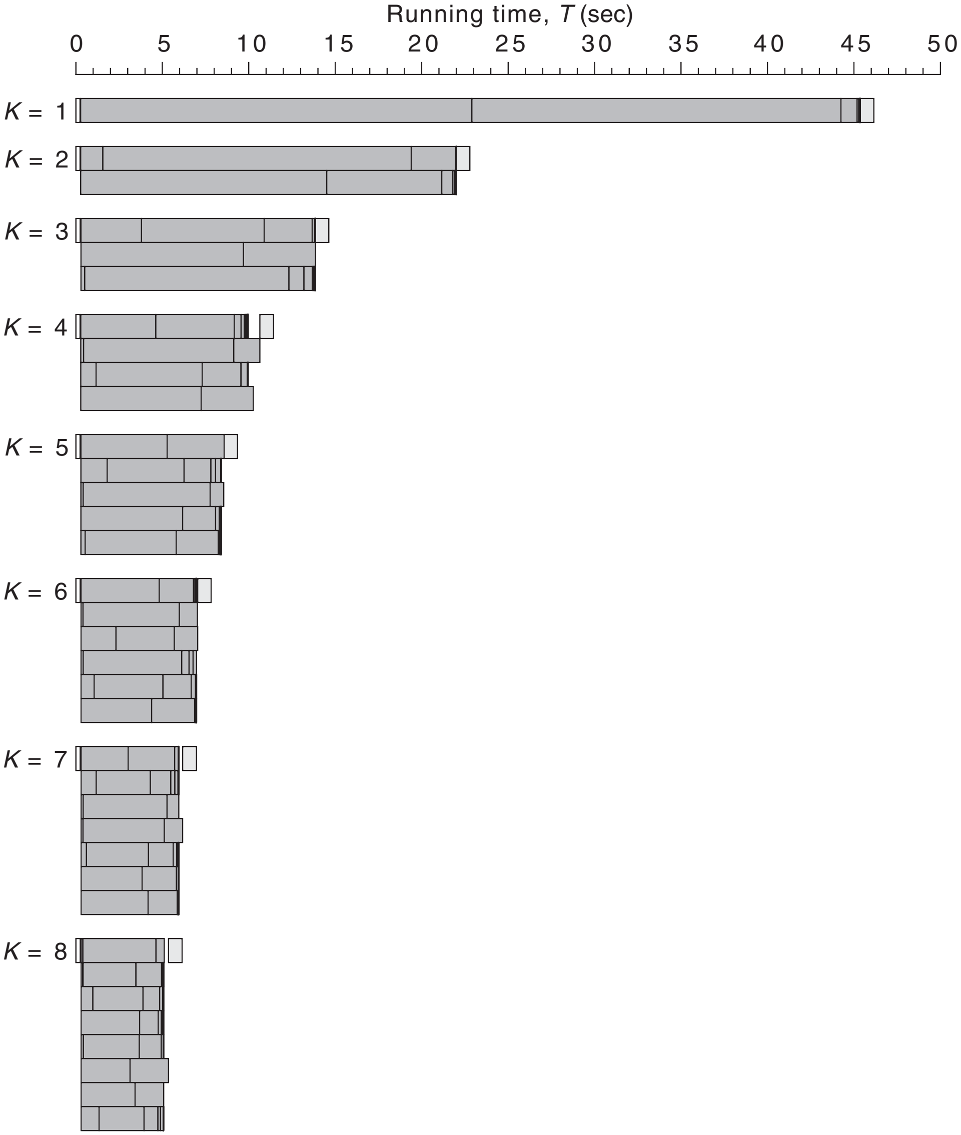


**Figure 12.8** Running times for each portion of the MandelbrotSetSmp program on different numbers for processors, guided schedule

The measured *B* values follow. They all are now about the same and all are close to 1, as they should be, indicating that the load stays balanced as the number of processors increases.

| $K$ | $T_p(K)$ | $T_p(1)$ | $B$ |
|---|---|---|---|
| 2 | 21778 | 45123 | 0.97 |
| 3 | 13607 | 45123 | 0.90 |
| 4 | 10401 | 45123 | 0.92 |
| 5 | 8332 | 45123 | 0.92 |
| 6 | 6796 | 45123 | 0.90 |
| 7 | 5926 | 45123 | 0.92 |
| 8 | 5103 | 45123 | 0.90 |

In fact, the *B* values are less than 1, which seems to indicate that the load balance is "better than perfect." Also, the efficiencies are mostly greater than 1, which seems to indicate that the speedups and efficiencies are "better than ideal." This behavior is due to the JVM's just-in-time (JIT) compiler. To gain some insight into the JIT compiler's effect on the program's performance, the program's running time was measured for $N$=10M pixels with the JIT compiler turned off. Table 12.2 (at the end of the chapter) lists the running time data, and Figure 12.9 plots the data with and without the JIT compiler. As we saw in Chapter 9, without the JIT compiler, the program runs about 30 times slower. Without the JIT compiler, the speedups are less than $K$, the efficiencies are less than 1, and the *EDSFs* are closer to constant, as we expect from Amdahl's Law. So the JIT compiler is indeed responsible for the odd speedup, efficiency, and *EDSF* curves.

How does the JIT compiler pull off this feat? With a guided schedule, the more threads there are, the more chunks there are in the parallel for loop, thus the more calls there are to the parallel for loop's `run()` method. With multiple threads executing more `run()` method calls, the JVM can detect that the `run()` method is a hot spot sooner than when a single thread is running. This, in turn, lets the JVM compile the `run()` method to machine code sooner and lets more of the program execute in the faster machine code mode when there are multiple threads. This, in turn, reduces the parallel portion's running time even further than would happen merely by switching to more threads. So $T_{\text{par}}(N,K)$ turns out smaller than $T_{\text{seq}}(N,1)/K$, the speedup ends up greater than $K$, and the efficiency ends up greater than 1. We will observe this same **JIT compiler effect** in other parallel programs.

Figure 12.7 plots the Mandelbrot Set program's *EDSF* versus $K$. (Most of the values are negative due to the JIT compiler effect.) Note that, despite fluctuations due to measurement error, the *EDSF* increases as the number of processors increases. This increase also is due to the JIT compiler. After compiling the Java bytecode to machine code, the JVM continues to optimize the compiled machine code, and the longer the program runs, the more optimizations the JVM applies. When the program runs for a short time, the JVM does not get the chance to optimize the machine code fully. Thus, with a small number of threads, the program runs for a longer period, the JVM optimizes the machine code more, the program's speed is larger, and the measured *EDSF* is smaller. With a large number of threads, the program runs for a shorter period, the JVM does not optimize the machine code as much, the program's speed is smaller, and the measured *EDSF* is larger.

To sum up, now that we've dealt with the load balancing, the speedup and efficiency curves look more like Amdahl's Law says they should. As *K* increases, the efficiencies increase above 1 due to the JIT compiler effect, and then the efficiencies drop off again due to the program's sequential fraction. However, because this program's sequential fraction is small, the efficiency reduction is minor. Most of the program's sequential time is occupied compressing the image's pixel data and writing the PJG file. PJG's fast compression algorithm ensures that the program's sequential time is small relative to its parallel time; this is the reason we went with PJG in the first place.

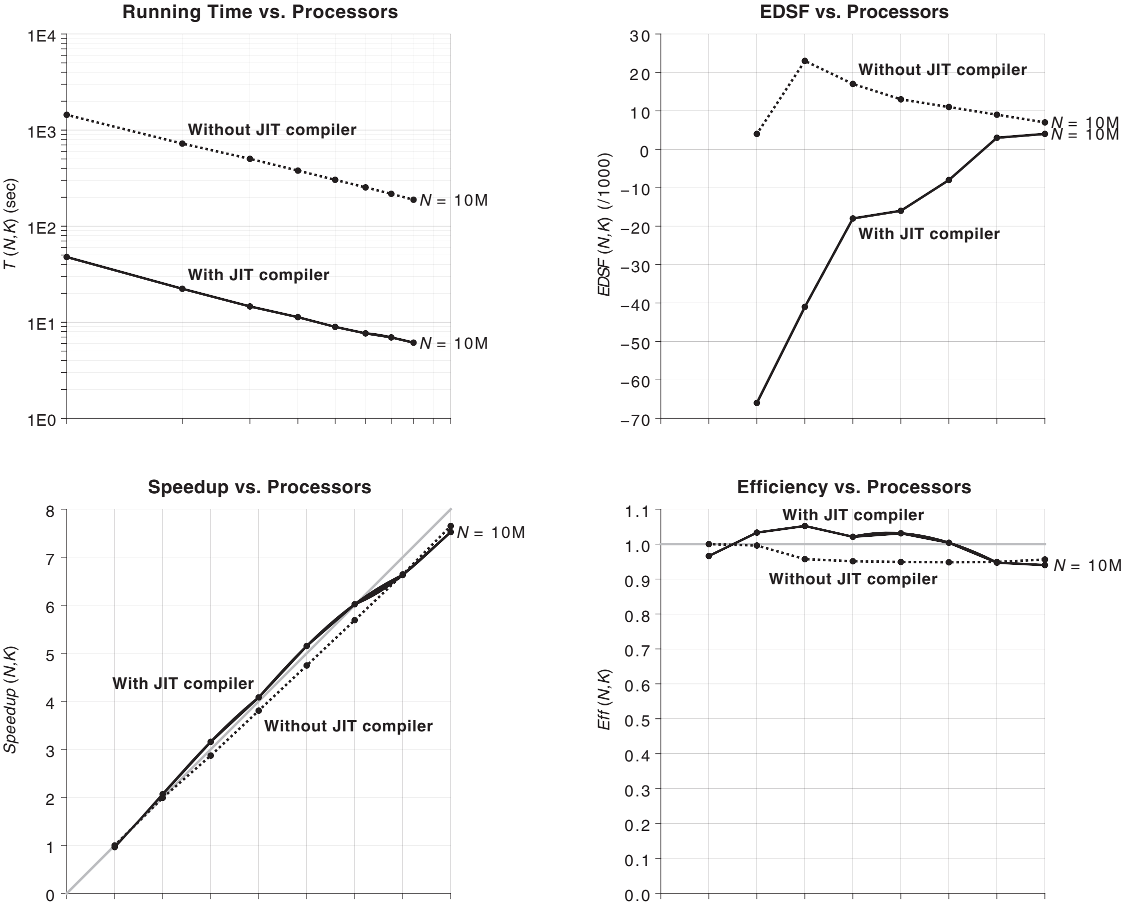


**Figure 12.9** MandelbrotSetSeq/MandelbrotSetSmp running-time metrics, guided schedule, with and without JIT compiler

## 12.5 For Further Information

On the Mandelbrot Set and many other aspects of fractals:

- B. Mandelbrot. *The Fractal Geometry of Nature*. W. H. Freeman and Company, 1977.

**Table 12.1** MandelbrotSetSeq/MandelbrotSetSmp running time metrics, guided schedule

| *N* | *K* | *T* | *Spdup* | *Eff* | *EDSF* | *N* | *K* | *T* | *Spdup* | *Eff* | *EDSF* |
|---:|---:|---:|---:|---:|---:|---:|---:|---:|---:|---:|---:|
| 10M | seq | 46082 | | | | 80M | seq | 358869 | | | |
| 10M | 1 | 47725 | 0.966 | 0.966 | | 80M | 1 | 358904 | 1.000 | 1.000 | |
| 10M | 2 | 22295 | 2.067 | 1.033 | -0.066 | 80M | 2 | 170638 | 2.103 | 1.052 | -0.049 |
| 10M | 3 | 14597 | 3.157 | 1.052 | -0.041 | 80M | 3 | 110982 | 3.234 | 1.078 | -0.036 |
| 10M | 4 | 11288 | 4.082 | 1.021 | -0.018 | 80M | 4 | 86138 | 4.166 | 1.042 | -0.013 |
| 10M | 5 | 8943 | 5.153 | 1.031 | -0.016 | 80M | 5 | 66782 | 5.374 | 1.075 | -0.017 |
| 10M | 6 | 7653 | 6.021 | 1.004 | -0.008 | 80M | 6 | 56540 | 6.347 | 1.058 | -0.011 |
| 10M | 7 | 6950 | 6.631 | 0.947 | 0.003 | 80M | 7 | 48342 | 7.424 | 1.061 | -0.010 |
| 10M | 8 | 6125 | 7.524 | 0.940 | 0.004 | 80M | 8 | 44695 | 8.029 | 1.004 | -0.001 |
| 20M | seq | 89953 | | | | 160M | seq | 732332 | | | |
| 20M | 1 | 92936 | 0.968 | 0.968 | | 160M | 1 | 730873 | 1.002 | 1.002 | |
| 20M | 2 | 42883 | 2.098 | 1.049 | -0.077 | 160M | 2 | 347837 | 2.105 | 1.053 | -0.048 |
| 20M | 3 | 27993 | 3.213 | 1.071 | -0.048 | 160M | 3 | 223581 | 3.275 | 1.092 | -0.041 |
| 20M | 4 | 21804 | 4.126 | 1.031 | -0.021 | 160M | 4 | 166094 | 4.409 | 1.102 | -0.030 |
| 20M | 5 | 16934 | 5.312 | 1.062 | -0.022 | 160M | 5 | 136216 | 5.376 | 1.075 | -0.017 |
| 20M | 6 | 14371 | 6.259 | 1.043 | -0.014 | 160M | 6 | 118190 | 6.196 | 1.033 | -0.006 |
| 20M | 7 | 12316 | 7.304 | 1.043 | -0.012 | 160M | 7 | 100780 | 7.267 | 1.038 | -0.006 |
| 20M | 8 | 11337 | 7.934 | 0.992 | -0.003 | 160M | 8 | 90421 | 8.099 | 1.012 | -0.001 |
| 40M | seq | 183295 | | | | 320M | seq | 1433954 | | | |
| 40M | 1 | 183093 | 1.001 | 1.001 | | 320M | 1 | 1481640 | 0.968 | 0.968 | |
| 40M | 2 | 87317 | 2.099 | 1.050 | -0.046 | 320M | 2 | 680360 | 2.108 | 1.054 | -0.082 |
| 40M | 3 | 56874 | 3.223 | 1.074 | -0.034 | 320M | 3 | 442219 | 3.243 | 1.081 | -0.052 |
| 40M | 4 | 44096 | 4.157 | 1.039 | -0.012 | 320M | 4 | 324784 | 4.415 | 1.104 | -0.041 |
| 40M | 5 | 34311 | 5.342 | 1.068 | -0.016 | 320M | 5 | 265607 | 5.399 | 1.080 | -0.026 |
| 40M | 6 | 29032 | 6.314 | 1.052 | -0.010 | 320M | 6 | 224330 | 6.392 | 1.065 | -0.018 |
| 40M | 7 | 24893 | 7.363 | 1.052 | -0.008 | 320M | 7 | 196165 | 7.310 | 1.044 | -0.012 |
| 40M | 8 | 22860 | 8.018 | 1.002 | 0.000 | 320M | 8 | 176553 | 8.122 | 1.015 | -0.007 |

**Table 12.2** MandelbrotSetSeq/MandelbrotSetSmp running time metrics, guided schedule, without JIT compiler

| *N* | *K* | *T* | *Spdup* | *Eff* | *EDSF* |
|---:|---:|---:|---:|---:|---:|
| 10M | seq | 1440962 | | | |
| 10M | 1 | 1440995 | 1.000 | 1.000 | |
| 10M | 2 | 723308 | 1.992 | 0.996 | 0.004 |
| 10M | 3 | 502099 | 2.870 | 0.957 | 0.023 |
| 10M | 4 | 378627 | 3.806 | 0.951 | 0.017 |
| 10M | 5 | 303613 | 4.746 | 0.949 | 0.013 |
| 10M | 6 | 253245 | 5.690 | 0.948 | 0.011 |
| 10M | 7 | 216955 | 6.642 | 0.949 | 0.009 |
| 10M | 8 | 188342 | 7.651 | 0.956 | 0.007 |

# CHAPTER 13 Reduction

in which we study a program where the threads must synchronize with each other when accessing shared variables; we discover how this synchronization can severely reduce the program's performance; and we learn how to overcome the problem using the parallel reduction pattern

# CHAPTER 13 Reduction

## 13.1 Estimating pi Using Random Numbers

Imagine we have a square dartboard, with each side being 1 unit in length (Figure 13.1). We use a compass to draw a quadrant of a circle of radius 1 unit centered at the lower-left corner. We throw a large number of darts at the board. What fraction of the dart holes will lie within the circle quadrant?

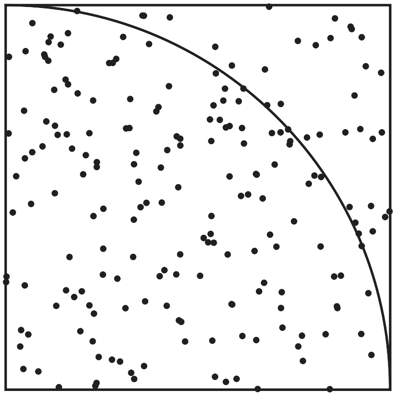


**Figure 13.1** A dartboard for estimating $\pi$

Suppose we count the dart holes and find that when we threw $N$ darts, $C$ of them landed within the circle quadrant. Assuming the darts landed at random locations spread uniformly across the dartboard, the ratio of the number of darts within the circle quadrant to the total number of darts, $C/N$, should be approximately the same as the ratio of the circle quadrant's area to the square's area. Because a full circle of radius $r$ has an area of $\pi r^2$, the circle quadrant's area is $\pi/4$. Because a square of side $r$ has an area of $r^2$, the square's area is 1. Therefore, $C/N$ should be approximately $\pi/4$.

This immediately suggests how a computer program can estimate the value of $\pi$. Generate a large number $N$ of points $(x,y)$, where $x$ and $y$ are each chosen uniformly at random in the range 0 to 1. Determine whether each point falls within the circle quadrant, that is, whether each point's distance from the origin is less than or equal to 1. (Actually, test the squared distance, $x^2 + y^2 \le 1^2$, to avoid unnecessary square root calculations.) Count the number of points $C$ that fall within the circle quadrant. Because $C/N$ is approximately $\pi/4$, print out $4C/N$ as the estimate for $\pi$.

Algorithms such as the preceding that calculate their results using random numbers are called **Monte Carlo algorithms**, after the famed casino of Monte Carlo (Figure 13.2), where random number generators—roulette wheels, dice, shuffled packs of cards—govern the outcome. This particular algorithm for estimating $\pi$ is an example of a **Monte Carlo integration** algorithm. Technically, we are calculating the integral—the area under the curve—of the mathematical function $y(x)$ for a circle of radius 1:

$$\int_0^1 \sqrt{1-x^2}\,\mathrm{d}x = \frac{\pi}{4} \tag{13.1}$$

The area under the curve, as a fraction of the unit square's area, is estimated by the number of random points under the curve as a fraction of the total number of points.

Courtesy of Berthold Werner. http://commons.wikimedia.org/wiki/Image:Casino_2005.jpg

**Figure 13.2** The Casino of Monte Carlo

The Monte Carlo algorithm yields an *estimate* of the integral's true value. The estimate's accuracy is proportional to the square root of the number of random points $N$. If we want to double the accuracy of the answer, we have to do four times as many random points; if we want one additional decimal place of accuracy, an improvement by a factor of 10, we have to do 100 times as many random points. Although Monte Carlo algorithms are easy to program, to get reasonably accurate answers they require sampling massive numbers of random points. Thus, Monte Carlo algorithms are attractive candidates for parallel programs.

## 13.2 Sequential Program

To embody the Monte Carlo algorithm for $\pi$ in a program is straightforward. Here is the source code for class PiSeq, a sequential program.

```
package edu.rit.smp.monte;
import java.util.Random;
public class PiSeq
   {
   // Command line arguments.
   static long seed;
   static long N;
```

To generate random points, the program uses a **pseudorandom number generator (PRNG)**. (This is not a cryptographic program, so we don't need to use an entropy source.) For now we will use an instance of class java.util.Random, the standard PRNG class in the Java platform.

```
   // Pseudorandom number generator.
   static Random prng;

   // Number of points within the unit circle.
   static long count;

   /**
    * Main program.
    */
   public static void main
      (String[] args)
      throws Exception
      {
      // Start timing.
      long time = -System.currentTimeMillis();
```

The program's command-line arguments include a **seed** for the PRNG and the number of points to generate, *N*. When using a PRNG, we always specify a seed. Initializing the PRNG with the same seed causes the PRNG to generate the same sequence of random numbers. This lets us run the program many times and get the same output. We'll use type `long` for *N* and for the counter; this lets us do up to $2^{63} - 1$ iterations. (An `int` can go up to only $2^{31} - 1$ iterations, or about two billion.)

```
      // Parse command line arguments.
      if (args.length != 2) usage();
      seed = Long.parseLong (args[0]);
      N = Long.parseLong (args[1]);

      // Set up PRNG.
      prng = new Random (seed);
```

The computational heart of the program loops *N* times. On each iteration, it uses the PRNG to generate two random double-precision floating-point numbers *x* and *y*, tests whether (*x,y*) falls within the unit circle, and, if so, increments the counter.

```
      // Generate n random points in the unit square, count how
      // many are in the unit circle.
      count = 0;
      for (long i = 0; i < N; ++ i)
         {
         double x = prng.nextDouble();
         double y = prng.nextDouble();
         if (x*x + y*y <= 1.0) ++ count;
         }

      // Stop timing.
      time += System.currentTimeMillis();

      // Print results.
      System.out.println
         ("pi = 4 * " + count + " / " + N + " = " +
          (4.0 * count / N));
      System.out.println (time + " msec");
      }
   }
```

Here is what the program printed for *seed* = 142857 and *N* = 10 million.

```
$ java edu.rit.smp.monte.PiSeq 142857 10000000
pi = 4 * 7854789 / 10000000 = 3.1419156
2746 msec
```

## 13.3 Parallel Program

To convert the sequential program to an SMP parallel program is also, seemingly, straightforward. We replace the sequential loop with a parallel loop so that multiple threads can throw darts at once. But if we do that, the `prng` variable and the `count` variable become *shared* variables, and we must consider whether the threads must synchronize with each other when accessing these variables.

Each `prng.nextDouble()` method call is an update operation. The calling thread reads the PRNG's state from a private field of the Random object, generates a random value as the PRNG's new state, and writes the new state back into the private field. Thus, multiple threads calling `prng.nextDouble()` must synchronize with each other.

Anticipating this, the designers of class Random chose to make the class **multiple-thread safe**. They did so by using an **atomic compare-and-set (CAS)** operation to update the private field that holds the PRNG's state. When one thread tries to update the private field, the CAS operation checks whether there was a conflict with another thread updating the private field at the same time. If there was a conflict, the CAS operation does not perform the update, and the thread must retry the update. The thread must retry repeatedly, if necessary, until the update succeeds. (See Appendix D for a fuller explanation of how the CAS operation works.) The CAS operation ensures that multiple threads will not interfere with each other's updates, making class Random multiple-thread safe. That being the case, the parallel program can call methods on the `prng` variable in the same way as the sequential program.

Each increment of the `count` variable is also an update operation. The executing thread reads the variable's value, adds 1, and writes the new value back into the variable. Thus, multiple threads incrementing the `count` variable must also synchronize with each other.

While class Random is multiple-thread safe, type `long` is not. We no longer can use type `long` for the `count` variable in the multithreaded parallel program. Instead, we will use a multiple-thread-safe class from the Parallel Java Library, class edu.rit.pj.reduction.SharedLong. Class SharedLong has a private field of type `long` as well as methods for reading, writing, and updating the private field. Like class Random, class SharedLong achieves multiple-thread safety using an atomic CAS operation.

Here is the source code for class PiSmp, an SMP parallel version of the Monte Carlo $\pi$ program.

```
package edu.rit.smp.monte;
import edu.rit.pj.LongForLoop;
import edu.rit.pj.ParallelTeam;
import edu.rit.pj.ParallelRegion;
import edu.rit.pj.reduction.SharedLong;
import java.util.Random;
public class PiSmp
   {
   // Command line arguments.
   static long seed;
   static long N;

   // Pseudorandom number generator.
   static Random prng;
```

As mentioned earlier, the `count` variable is now an instance of class SharedLong.

```
// Number of points within the unit circle.
static SharedLong count;

/**
 * Main program.
 */
public static void main
   (String[] args)
   throws Exception
   {
   // Start timing.
   long time = -System.currentTimeMillis();

   // Validate command line arguments.
   if (args.length != 2) usage();
   seed = Long.parseLong (args[0]);
   N = Long.parseLong (args[1]);

   // Set up PRNG.
   prng = new Random (seed);

   // Generate n random points in the unit square, count how
   // many are in the unit circle.
   count = new SharedLong (0L);
   new ParallelTeam().execute (new ParallelRegion()
      {
      public void run() throws Exception
         {
```

The plain for loop in the sequential version becomes a parallel loop in the parallel version. Instead of class IntegerForLoop, we use class LongForLoop, whose loop index variable is type `long`.

```
         execute (0, N-1, new LongForLoop()
            {
            public void run (long first, long last)
               {
               for (long i = first; i <= last; ++ i)
                  {
```

To get a random number, each thread calls a method on the multiple-thread-safe `prng` object. To increment the counter, each thread calls a method on the multiple-thread-safe `count` object.

```
                  double x = prng.nextDouble();
                  double y = prng.nextDouble();
                  if (x*x + y*y <= 1.0) count.incrementAndGet();
                  }
               }
            });
         }
      });

   // Stop timing.
   time += System.currentTimeMillis();

   // Print results.
   System.out.println
      ("pi = 4 * " + count + " / " + N + " = " +
       (4.0 * count.doubleValue() / N));
   System.out.println (time + " msec");
   }
}
```

Don't be too eager to emulate the preceding parallel program. It has two serious design flaws. One flaw has to do with performance; the other flaw has to do with the way in which multiple threads generate random numbers.

The first flaw becomes apparent when we measure the PiSmp program's running time on the "parasite" SMP computer. Table 13.1 lists the running-time data for a problem size of $N$ = 200 million darts. Figure 13.3 plots the running time and speedup versus the number of processors. The parallel program's speedup on one processor, which ought to be very close to 1, instead is only 0.900. Much worse, the running time on two processors is *longer* than the running time on one processor, and the more processors we add, the longer the running time becomes. We are getting a *slowdown* instead of a speedup! Before doing anything else, we must determine the cause of the slowdown and fix the problem.

**Table 13.1** PiSeq/PiSmp running time metrics

| *N* | *K* | *T* | *Spdup* | *Eff* | *EDSF* |
|---:|---:|---:|---:|---:|---:|
| 200M | seq | 53167 | | | |
| 200M | 1 | 59105 | 0.900 | 0.900 | |
| 200M | 2 | 137653 | 0.386 | 0.193 | 3.658 |
| 200M | 3 | 212120 | 0.251 | 0.084 | 4.883 |
| 200M | 4 | 319983 | 0.166 | 0.042 | 6.885 |
| 200M | 5 | 349657 | 0.152 | 0.030 | 7.145 |
| 200M | 6 | 451591 | 0.118 | 0.020 | 8.969 |
| 200M | 7 | 539920 | 0.098 | 0.014 | 10.491 |
| 200M | 8 | 636163 | 0.086 | 0.010 | 12.158 |

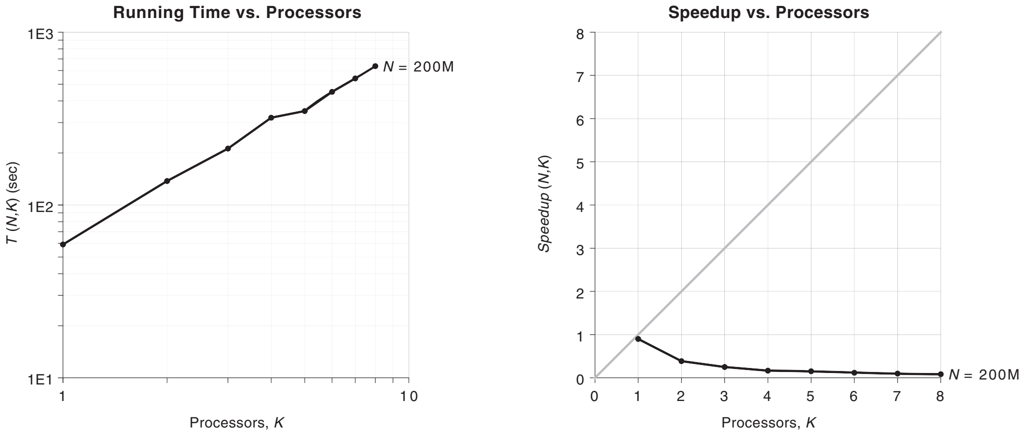


**Figure 13.3** PiSeq/PiSmp running time metrics

## 13.4 The Reduction Pattern

Ironically, the PiSmp program's multiple-thread safety is what's causing the slowdown.

Consider how the program uses the shared PRNG (Figure 13.4). Because class Random is multiple-thread safe, only one thread at a time is allowed to get a random number from the PRNG. But the threads are doing almost nothing except getting random numbers. Consequently, each thread spends much of its time waiting its turn to access the PRNG instead of computing. In addition, the parallel program must spend some time doing an atomic CAS operation each time it increments the `count` variable, even if there's only one thread. This extra overhead, not present in the sequential program, causes the parallel program's running time on one processor to exceed the sequential program's running time, resulting in a speedup of only 0.900. Worse, the more threads that are trying to do atomic CAS operations on the `prng` and `count` variables, the more likely that conflicts will occur, and the more retries that each thread must do. Thus, the time to get a random number and to increment the counter becomes longer the more threads there are.

Now consider that this thread-synchronization overhead happens on *each and every loop iteration;* and to get accurate answers from the Monte Carlo algorithm, we will need *millions or billions* of loop iterations. It's little wonder the program slows down as we add more processors (threads).

We are facing a dilemma. If we leave the thread synchronization out, the threads will interfere with each other when accessing the shared variables and the program will compute the wrong answer. If we put the thread synchronization in, the program's performance goes way down.

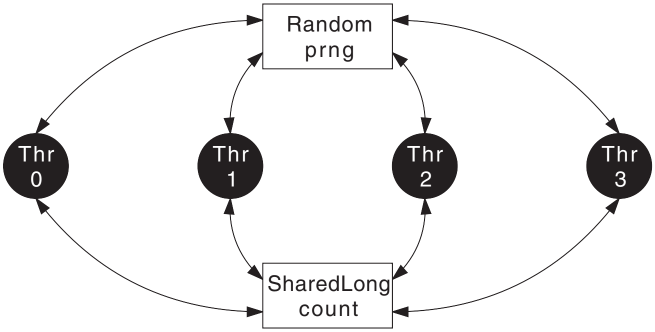


**Figure 13.4** Threads accessing variables

The way out of the dilemma is for the threads not to contend with each other—not to access shared variables—while doing their loop iterations. Instead, each thread will have its own *per-thread* PRNG variable and *per-thread* counter variable (Figure 13.5). Each thread can then update its own PRNG and counter without interference from the other threads and without having to contend with the other threads for access to shared variables. Once the threads have finished their iterations, the per-thread counters must be added together to yield the final total count. Because all the threads must update the same total count, the total counter variable must still be a shared variable; and because multiple threads are updating the total count, the shared variable must still be multiple-thread safe. However, this time, each thread has to synchronize with the other threads only *once* at the end of the program, not on every loop iteration. Hence, the program's performance should be a lot better.

When multiple individual per-thread results are combined, or reduced, in this way to yield an overall result, the program is following the **parallel reduction pattern**. The shared variable that holds the combined result is called a **reduction variable**. The classes in package edu.rit.pj.reduction in the Parallel Java Library are designed to be used for shared reduction variables in SMP parallel programs. Package edu.rit.pj.reduction provides multiple-thread-safe wrapper classes for each primitive type as well as object types, and for arrays thereof.

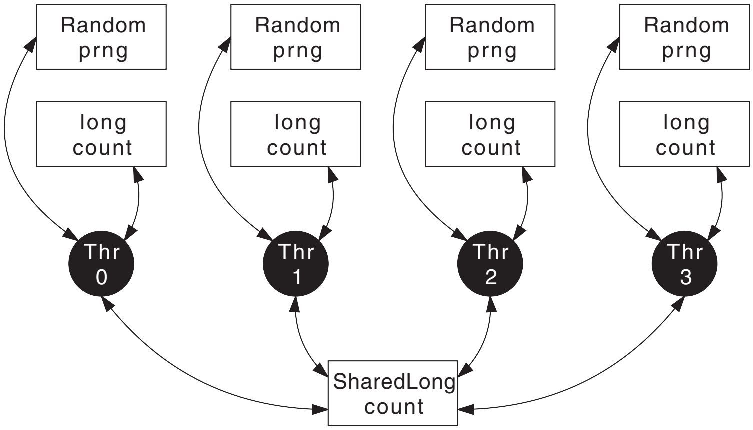


**Figure 13.5** Threads accessing variables using the reduction pattern

## 13.5 Parallel Program with Reduction

Here is the source code for class PiSmp2, an SMP parallel version of the Monte Carlo $\pi$ program that uses parallel reduction.

```
package edu.rit.smp.monte;
import edu.rit.pj.LongForLoop;
import edu.rit.pj.ParallelTeam;
import edu.rit.pj.ParallelRegion;
import edu.rit.pj.reduction.SharedLong;
import java.util.Random;
public class PiSmp2
   {
   // Command line arguments.
   static long seed;
   static long N;
```

The shared PRNG variable is gone. The shared counter variable is still here, but now it is used as a reduction variable.

```
   // Number of points within the unit circle.
   static SharedLong count;

   /**
    * Main program.
    */
   public static void main
      (String[] args)
      throws Exception
      {
      // Start timing.
      long time = -System.currentTimeMillis();

      // Validate command line arguments.
      if (args.length != 2) usage();
      seed = Long.parseLong (args[0]);
      N = Long.parseLong (args[1]);

      // Generate n random points in the unit square, count how
      // many are in the unit circle.
      count = new SharedLong (0L);
      new ParallelTeam().execute (new ParallelRegion()
         {
         public void run() throws Exception
```

```
{
execute (0, N-1, new LongForLoop()
   {
```

Now each thread has its own PRNG and counter. Because these are fields of the LongForLoop subclass, each thread gets its own copies when it executes the loop. Because only one thread accesses these variables, we can go back to the non-multiple-thread-safe type `long` for the counter. It takes less time to increment a `long` than to increment a SharedLong because the former does not need to do an atomic compare-and-set operation. (Note the extra padding to avert cache interference when the threads access their own per-thread variables.)

```
   // Set up per-thread PRNG and counter.
   Random prng_thread = new Random (seed);
   long count_thread = 0;

   // Extra padding to avert cache interference.
   long pad0, pad1, pad2, pad3, pad4, pad5, pad6, pad7;
   long pad8, pad9, pada, padb, padc, padd, pade, padf;

   // Parallel loop body.
   public void run (long first, long last)
      {
      // Generate random points.
      for (long i = first; i <= last; ++ i)
         {
```

Inside the loop, each thread now accesses its own PRNG and counter.

```
         double x = prng_thread.nextDouble();
         double y = prng_thread.nextDouble();
         if (x*x + y*y <= 1.0) ++ count_thread;
         }
      }
```

After finishing the loop, each thread performs the parallel reduction by adding its own per-thread counter into the shared reduction variable. Putting this code in the LongForLoop's `finish()` method ensures that each thread will run this code only once, no matter how many chunks of loop iterations the thread executes (as dictated by the loop schedule).

```
            public void finish()
               {
               // Reduce per-thread counts into shared count.
               count.addAndGet (count_thread);
               }
            });
         }
      });

   // Stop timing.
   time += System.currentTimeMillis();

   // Print results.
   System.out.println
      ("pi = 4 * " + count + " / " + N + " = " +
       (4.0 * count.doubleValue() / N));
   System.out.println (time + " msec");
   }
}
```

Table 13.2 (at the end of the chapter) gives the running-time measurements in milliseconds for the Monte Carlo $\pi$ program with parallel reduction for various problem sizes *N* from 200 million (200M, for “mega”) to 10 billion (10G, for “giga”) darts, as well as the speedups, efficiencies, and *EDSFs* calculated from the running times. Figure 13.6 plots the running-time metrics versus the number of processors. Switching to the parallel reduction pattern has eradicated the slowdowns, and we are now seeing excellent speedups and efficiencies.

The *EDSF* curves are curious. There is an abrupt increase in the *EDSF* values when going from $K = 4$ processors to $K = 5$ processors. This behavior is due to the CPU hardware. The “parasite” SMP parallel computer has four Sun Microsystems UltraSPARC-IV CPU chips. These are hyperthreaded CPUs, each having two instruction units that can issue instructions for two threads simultaneously, but each having only one set of functional units to execute instructions. For $K = 1$ to 4, the JVM and operating system schedule each parallel team thread on a different CPU chip, and each thread has the whole CPU to itself. For $K = 5$ to 8, the JVM and operating system start scheduling two threads per CPU chip, and the threads have to share the CPU’s functional units. Because the threads are executing the same instructions, the threads are contending with each other to use the same functional units, and sometimes one thread must wait for the other thread to finish using a particular functional unit. These waits cause the program’s *EDSF* to increase. These waits also cause the program’s speedup and efficiency to dip slightly as we go from $K = 4$ to 5.

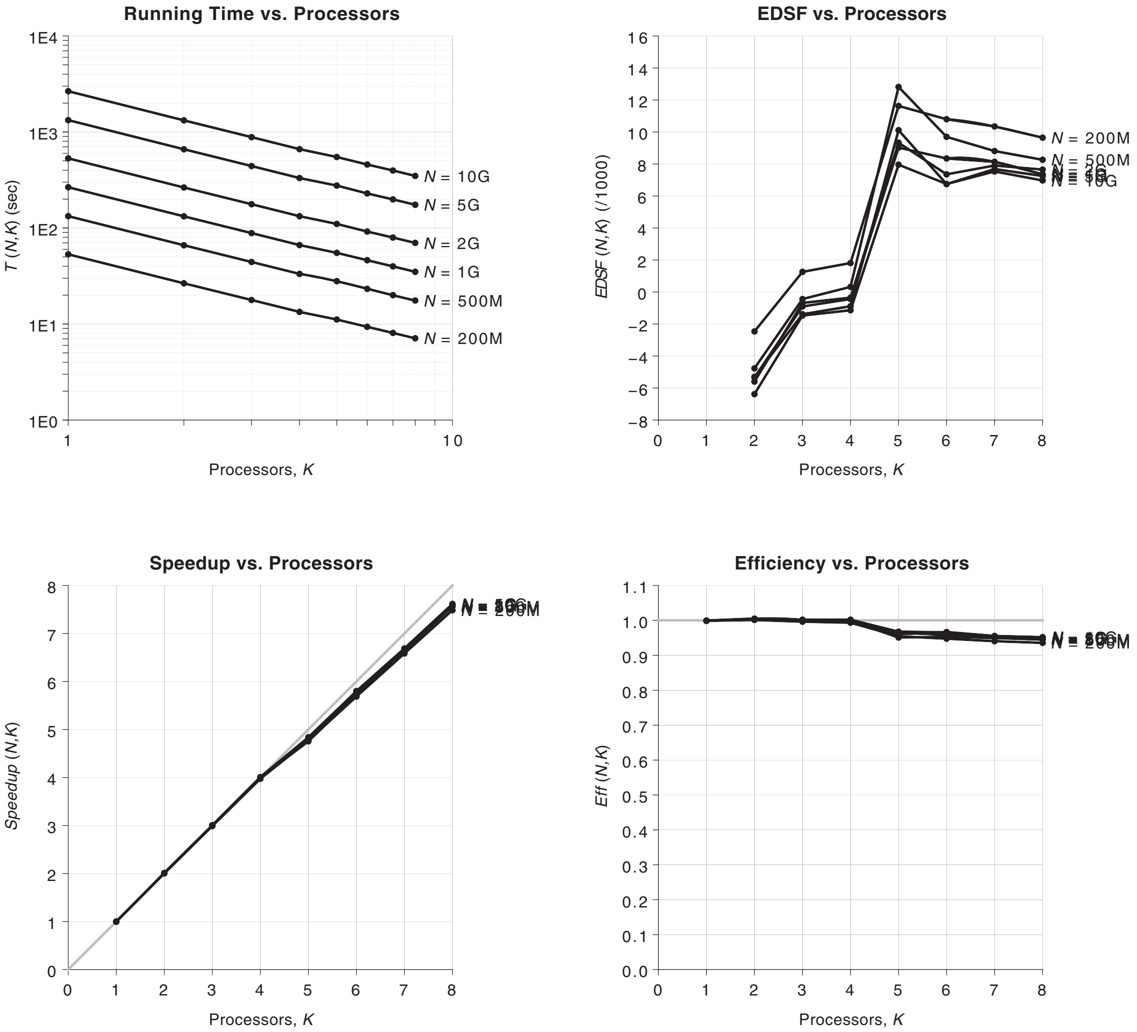


**Figure 13.6** PiSeq/PiSmp2 running-time metrics

## 13.6 The Second Flaw

Now that we've fixed the first design flaw in the Monte Carlo $\pi$ program and are getting decent performance, we turn our attention to the second design flaw. This one is more subtle. Here are the values of $\pi$ the program calculated with a seed of 142857 and with $N = 200$ million darts running on different numbers of processors $K$.

| $K$ | $\pi$ |
|---|---|
| seq | 3.14166696 |
| 1 | 3.14166696 |
| 2 | 3.14177916 |
| 3 | 3.14187478 |
| 4 | 3.14195848 |
| 5 | 3.14187660 |
| 6 | 3.14197380 |
| 7 | 3.14193412 |
| 8 | 3.14188656 |

The parallel version with one thread computed the same answer as the sequential version. But with more than one thread, the parallel version computed different answers! Although all the answers are within 0.01% of the true value of $\pi$, the program ought to be capable of giving identical answers no matter how many processors we use. Why are the answers all different?

In the parallel version with one thread, there is one PRNG. Initialized with the same seed as the sequential version, the PRNG generates the same sequence of random numbers. Thus, the parallel version generates 200 million random points $(x,y)$ that are the same as the ones the sequential version generates, so the parallel version with one thread computes the same answer as the sequential version.

In the parallel version with two threads, there are two PRNGs, each initialized with the same seed as the sequential version. Each thread in the parallel version does 100 million iterations. But this time, instead of generating 200 million different points, the parallel version generates 100 million different points, twice each—the same as the first 100 million points the sequential version generates. It is as though two identical twins were throwing darts at the dartboard—twins so identical that their darts always land in exactly the same place. The missing 100 million points explain why the parallel version with two (or more) threads computes an answer that differs from the sequential version.

We need to fix the parallel version so it gives the same answer as the sequential version, no matter how many processors it runs on. Before we can do so, we need to understand how PRNGs generate random numbers. That's the topic of the next chapter.

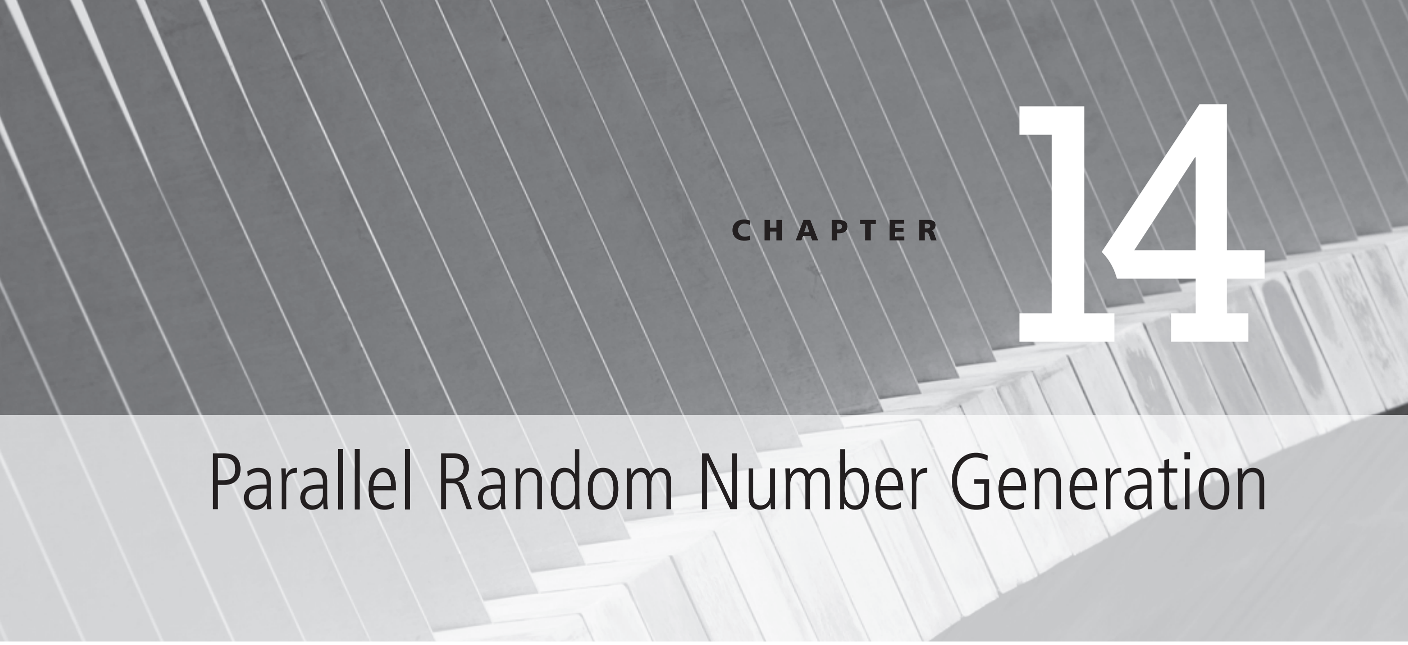
## 13.7 For Further Information

On Monte Carlo integration techniques:

- W. Press, S. Teukolsky, W. Vetterling, and B. Flannery. *Numerical Recipes: The Art of Scientific Computing, Third Edition*. Cambridge University Press, 2008, Chapter 7.

**Table 13.2** PiSeq/PiSmp2 running-time metrics

| *N* | *K* | *T* | *Spdup* | *Eff* | *EDSF* | *N* | *K* | *T* | *Spdup* | *Eff* | *EDSF* |
|---:|---:|---:|---:|---:|---:|---:|---:|---:|---:|---:|---:|
| 200M | seq | 53156 | | | | 2G | seq | 530894 | | | |
| 200M | 1 | 53215 | 0.999 | 0.999 | | 2G | 1 | 531417 | 0.999 | 0.999 | |
| 200M | 2 | 26542 | 2.003 | 1.001 | -0.002 | 2G | 2 | 264221 | 2.009 | 1.005 | -0.006 |
| 200M | 3 | 17783 | 2.989 | 0.996 | 0.001 | 2G | 3 | 176897 | 3.001 | 1.000 | -0.001 |
| 200M | 4 | 13376 | 3.974 | 0.993 | 0.002 | 2G | 4 | 132714 | 4.000 | 1.000 | 0.000 |
| 200M | 5 | 11138 | 4.772 | 0.954 | 0.012 | 2G | 5 | 110249 | 4.815 | 0.963 | 0.009 |
| 200M | 6 | 9348 | 5.686 | 0.948 | 0.011 | 2G | 6 | 91826 | 5.782 | 0.964 | 0.007 |
| 200M | 7 | 8074 | 6.584 | 0.941 | 0.010 | 2G | 7 | 79517 | 6.676 | 0.954 | 0.008 |
| 200M | 8 | 7101 | 7.486 | 0.936 | 0.010 | 2G | 8 | 69985 | 7.586 | 0.948 | 0.008 |
| 500M | seq | 132867 | | | | 5G | seq | 1328267 | | | |
| 500M | 1 | 132912 | 1.000 | 1.000 | | 5G | 1 | 1329193 | 0.999 | 0.999 | |
| 500M | 2 | 66139 | 2.009 | 1.004 | -0.005 | 5G | 2 | 661072 | 2.009 | 1.005 | -0.005 |
| 500M | 3 | 44265 | 3.002 | 1.001 | 0.000 | 5G | 3 | 441836 | 3.006 | 1.002 | -0.001 |
| 500M | 4 | 33260 | 3.995 | 0.999 | 0.000 | 5G | 4 | 331410 | 4.008 | 1.002 | -0.001 |
| 500M | 5 | 27945 | 4.755 | 0.951 | 0.013 | 5G | 5 | 276593 | 4.802 | 0.960 | 0.010 |
| 500M | 6 | 23226 | 5.721 | 0.953 | 0.010 | 5G | 6 | 229002 | 5.800 | 0.967 | 0.007 |
| 500M | 7 | 19991 | 6.646 | 0.949 | 0.009 | 5G | 7 | 198640 | 6.687 | 0.955 | 0.008 |
| 500M | 8 | 17575 | 7.560 | 0.945 | 0.008 | 5G | 8 | 174579 | 7.608 | 0.951 | 0.007 |
| 1G | seq | 265748 | | | | 10G | seq | 2655504 | | | |
| 1G | 1 | 265954 | 0.999 | 0.999 | | 10G | 1 | 2659653 | 0.998 | 0.998 | |
| 1G | 2 | 132267 | 2.009 | 1.005 | -0.005 | 10G | 2 | 1321340 | 2.010 | 1.005 | -0.006 |
| 1G | 3 | 88491 | 3.003 | 1.001 | -0.001 | 10G | 3 | 883942 | 3.004 | 1.001 | -0.001 |
| 1G | 4 | 66401 | 4.002 | 1.001 | 0.000 | 10G | 4 | 662640 | 4.007 | 1.002 | -0.001 |
| 1G | 5 | 55111 | 4.822 | 0.964 | 0.009 | 10G | 5 | 548864 | 4.838 | 0.968 | 0.008 |
| 1G | 6 | 46175 | 5.755 | 0.959 | 0.008 | 10G | 6 | 458232 | 5.795 | 0.966 | 0.007 |
| 1G | 7 | 39848 | 6.669 | 0.953 | 0.008 | 10G | 7 | 397122 | 6.687 | 0.955 | 0.008 |
| 1G | 8 | 34956 | 7.602 | 0.950 | 0.007 | 10G | 8 | 348678 | 7.616 | 0.952 | 0.007 |



# CHAPTER 14

# Parallel Random Number Generation

in which we look at patterns of random number generation in parallel programs; we learn how pseudorandom number generators work; we design a parallel pseudorandom number generator; and we apply it to our parallel Monte Carlo program

# CHAPTER 14 Parallel Random Number Generation

## 14.1 Parallel PRNG Patterns

The black-box interface to a pseudorandom generator (PRNG) is simple and standard: initialize the PRNG by setting the **seed**, and then repeatedly call some kind of `next()` method to generate a sequence of random numbers. A PRNG might have different `next()` methods that return numbers of various types, such as floating-point numbers between 0.0 and 1.0, or integers between 0 and a given upper bound. Because the set of random numbers a PRNG can generate is finite, after some number of random numbers have been generated, the PRNG repeats the same sequence of random numbers. That is, the PRNG goes through a **cycle** of random numbers; the initial seed value determines where in this cycle the PRNG starts. If you re-initialize the PRNG with the same seed, you will get the same sequence of random numbers.

The sequential Monte Carlo $\pi$ program from Chapter 13 uses a single PRNG to generate a certain sequence of random numbers (Figure 14.1). The white circle stands for the initial seed value, the black circles stand for the subsequently generated random numbers. The particular sequence depends on the initial seed value supplied as a command-line argument.

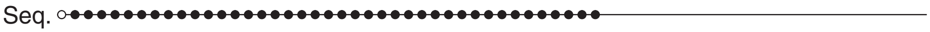


**Figure 14.1** Sequential random number sequence

The parallel Monte Carlo $\pi$ program from Chapter 13 uses a separate PRNG for each thread. But because each PRNG is initialized—incorrectly—with the same seed, each PRNG generates the identical random number sequence (Figure 14.2).

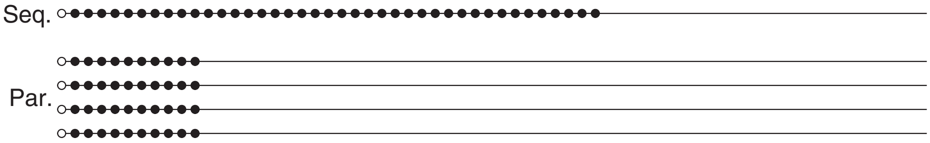


**Figure 14.2** Parallel random number sequences, incorrect

As we saw in Chapter 13, this causes the parallel program's output to differ from the sequential program's. To fix this problem, we can use any of three techniques: independent sequences, leapfrogging, or sequence splitting.

**Independent sequences**. Instead of initializing each thread's PRNG with the same seed, we can initialize them all with different seeds. This causes each PRNG to start at a different point in its cycle, and each PRNG will generate a different random number sequence (Figure 14.3).

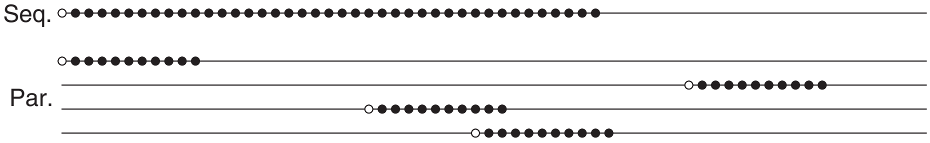


**Figure 14.3** Parallel random number sequences, independent sequences

However, there are several problems with this technique. First, if the parallel program has *K* threads, then *K* different seeds are needed instead of just one. Either the user must supply multiple seeds, or the program must somehow derive multiple seeds from a single user-supplied seed. Second, depending on the particular seeds chosen, it's possible that at some point in the program the sequence generated by one PRNG will overlap the sequence generated by another PRNG, as Figure 14.3 illustrates. We want to avoid overlapping sequences, because this might throw off the program's results. Finally, even if the PRNG's sequences do not overlap, it's unlikely that the set of random numbers generated by the parallel program's PRNGs together will be the same as the set of random numbers generated by the sequential program's single PRNG. This will prevent the parallel program from duplicating the sequential program's output.

**Leapfrogging**. After each thread initializes its PRNG with the same seed, thread 0 can leave its PRNG alone; thread 1 can tell its PRNG to *skip over* one random number; thread 2 can tell its PRNG to skip over two random numbers; and so on through thread *K*–1. When each thread now tells its PRNG to generate a random number, the numbers that come out will be the same as the first *K* numbers the sequential program's PRNG generated. Then, each thread tells its PRNG to skip over *K*–1 random numbers, namely the numbers the other PRNGs are generating. By repeatedly generating one random number and skipping *K*–1 random numbers, each thread *leapfrogs* over the other threads' random numbers (Figure 14.4).

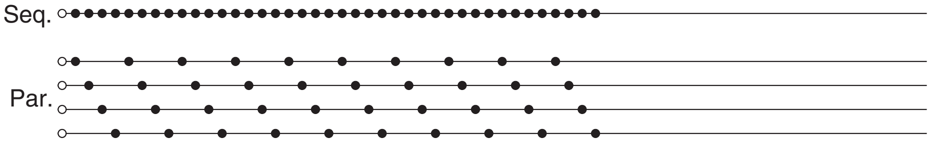


**Figure 14.4** Parallel random number sequences, leapfrogging

Unlike the independent sequences technique, with leapfrogging, the parallel program generates the same random numbers as the sequential program.

On each loop iteration, the Monte Carlo $\pi$ program generates a *pair* of random numbers. To get the same sequence of random points $(x,y)$ in the parallel version as in the sequential version using leapfrogging, each thread must repeatedly generate two random numbers and skip $2(K–1)$ random numbers (Figure 14.5).

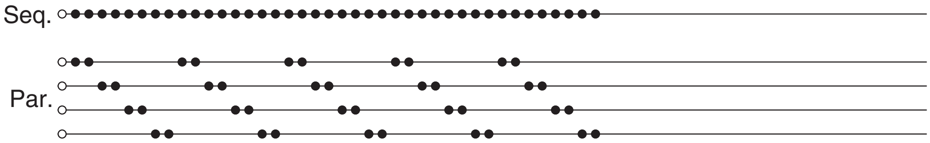


**Figure 14.5** Generating pairs of random numbers with leapfrogging

**Sequence splitting**. Suppose the program will generate *N* random numbers. After each thread initializes its PRNG with the same seed, thread 0 can leave its PRNG alone; thread 1 can tell its PRNG to skip *N*/*K* random numbers; thread 2 can tell its PRNG to skip 2*N*/*K* random numbers; and so on through thread *K*–1. Thereafter, each thread simply generates random numbers without any skipping (Figure 14.6).

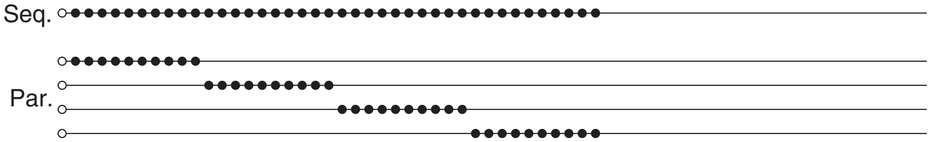


**Figure 14.6** Parallel random number sequences, sequence splitting

In effect, the random number sequence is split into *K* pieces, each with *N*/*K* consecutive random numbers. Like leapfrogging, with sequence splitting, the parallel program generates the same random numbers as the sequential program.

Which technique should you use? Keep in mind that skipping a PRNG an arbitrary distance ahead might take more time than generating the next random number. If you don't need the parallel program to generate the same random sequence when run on different numbers of processors, then the independent sequence technique is the easiest and fastest, because it does not involve any skipping. Otherwise, you must choose between leapfrogging and sequence splitting. Because leapfrogging does a skip after every random number and sequence splitting does only one skip at the beginning, a parallel program using sequence splitting might have a smaller running time than one using leapfrogging. On the other hand, sequence splitting requires that you know *N* (the number of random numbers needed) before you start generating random numbers; leapfrogging does not require knowing *N* ahead of time.

It is possible to do sequence splitting if you don't know *N* ahead of time. You can skip each PRNG's thread ahead a distance larger than any possible value of *N* (Figure 14.7). However, by doing so, the parallel program will not generate the same random numbers as the sequential program.

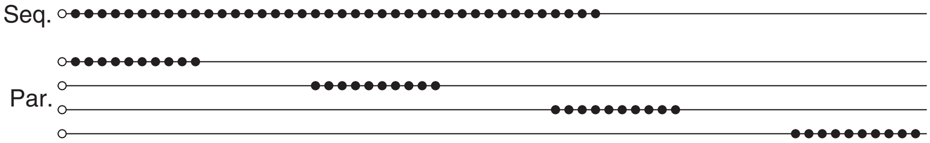


**Figure 14.7** Sequence splitting with *N* not known ahead of time

A key design criterion for a PRNG to be used in parallel programs, then, is that the PRNG algorithm supports an efficient skip operation. It must be possible to skip the PRNG an arbitrary number of positions ahead in its cycle much more quickly than calling the `next()` method that many times.

## 14.2 Pseudorandom Number Generator Algorithms

A PRNG algorithm has two aspects: a **hash function** and a **mode of operation**. The hash function, denoted *H*(*x*), takes an integer input value *x* and "randomizes" it, yielding a random number to be returned by the `next()` method. More precisely, the hash function maps each input value to a certain output value; for a given input value, the hash function always yields the same output value; but the mapping looks as if it were chosen at random. The mode of operation specifies how successive input values are presented to the hash function on successive `next()` method calls to yield a sequence of random numbers.

Many PRNG hash functions have been invented. We will look at four: the linear congruential generator, the multiplicative congruential generator, the xorshift generator, and the composite hash function. We will also look at two modes of operation: iterated mode and counter mode.

**Linear congruential generator (LCG)**. Discovered in the mid-twentieth century, the hash function for an LCG is

$$H(x) = x \cdot a + b \pmod{m} \tag{14.1}$$

where *a*, *b*, and *m* are constant parameters. We denote this hash function as *LCG*(*x*; *a*,*b*,*m*). It is called "linear" because it computes a linear function of *x*, and it is called "congruential" because the function's output is congruent to the linear function's value modulo *m*.

**Multiplicative congruential generator (MCG)**. An MCG is a special case of an LCG with *b* = 0,

$$H(x) = x \cdot a \pmod{m} \tag{14.2}$$

where *a* and *m* are constant parameters. We denote this hash function as *MCG*(*x*; *a*,*m*). The value *x* = 0 should be avoided, because in that case *H*(*x*) yields *x* (rather than a "random" output). Earlier, we said that the set of random numbers a PRNG can generate is finite. For an LCG or MCG, we now can see why. The (mod *m*) operation forces the output to be a member of the finite set {1, 2, …, *m*–1} (avoiding *x* = 0).

**Xorshift generator**. Invented by George Marsaglia in 2003, here is the pseudocode for the xorshift generator's hash function *H*(*x*),

$$\begin{array}{l} x \leftarrow x \;\oplus\; (x \gg a) \\ x \leftarrow x \;\oplus\; (x \ll b) \\ x \leftarrow x \;\oplus\; (x \gg c) \\ \text{Return } x \end{array} \tag{14.3}$$

where *a*, *b*, and *c* are constant parameters, ⊕ is the exclusive-or (xor) operation, ≫ is the right-shift operation, and ≪ is the left-shift operation. The operations are performed on *n*-bit unsigned integers, thus yielding outputs in the finite set {1, 2, …, $2^n$–1}. (Again, the value *x* = 0 should be avoided.) We denote this hash function as *XorShiftRight*(*x*; *a*,*b*,*c*).

A variation of the xorshift hash function *H*(*x*) begins with a left-shift instead of a right-shift:

$$\begin{aligned} &x \leftarrow x \,\oplus\, (x \ll a) \\ &x \leftarrow x \,\oplus\, (x \gg b) \\ &x \leftarrow x \,\oplus\, (x \ll c) \\ &\text{Return } x \end{aligned} \qquad (14.4)$$

We denote this hash function as *XorShiftLeft*(*x*; *a*,*b*,*c*).

**Composite hash function**. Feeding the output of one hash function into the input of another hash function gives still another hash function, a *composite* hash function:

$$H(x) = H_2(H_1(x)) \qquad (14.5)$$

Any number of hash functions can be composed together; for example, four:

$$H(x) = H_4(H_3(H_2(H_1(x)))) \qquad (14.6)$$

A composite hash function can be used to increase the degree of randomness when a single hash function is not random enough.

**Iterated mode**. Combining a hash function with a mode of operation gives a complete PRNG algorithm. In iterated mode, after setting the seed, the first random number generated is the hash of the seed; the second random number is the hash of the first random number; the third random number is the hash of the second random number; and so on. Treating the seed as an internal state variable, the `next()` method is the following:

$$\begin{aligned} &\mathit{seed} \leftarrow H(\mathit{seed}) \\ &\text{Return } \mathit{seed} \end{aligned} \qquad (14.7)$$

**Counter mode**. In this mode, the PRNG's internal state is a counter. Setting the seed initializes the counter. Successive generated random numbers are the hashes of successive counter values. The `next()` method is the following:

$$\begin{aligned} &\mathit{counter} \leftarrow \mathit{counter} + 1 \\ &\text{Return } H(\mathit{counter}) \end{aligned} \qquad (14.8)$$

Which mode of operation shall we use for our parallel PRNG algorithm? The PRNG must support an efficient skip operation. The obvious choice, then, is counter mode. To skip a counter-mode PRNG ahead *n* positions, simply increase the counter by *n* instead of 1. In general, there is no way to skip an iterated-mode PRNG ahead *n* positions other than by calling the `next()` method *n* times. While some hash functions do support faster skip operations—for example, an iterated-mode MCG can be skipped ahead *n* positions by a modular exponentiation using an *O*(log *n*) algorithm—these are still not as fast as adding *n* to a counter.

Another important design consideration is the PRNG's **period**. This is the length of the PRNG's cycle, the number of random numbers that can be generated before the sequence starts repeating itself. If a counter-mode PRNG's hash function maps each distinct input value to a different output value, the period will be $2^n$ for an $n$-bit counter. ($n$ is often chosen to be the computer's word size.) A 32-bit counter, however, is too small for use in large-scale parallel programs; the PRNG starts repeating itself after only four billion random numbers. A 64-bit counter is much better. Even then, the program should not consume more than a fraction of the PRNG's period. Otherwise, numbers generated later might correlate with numbers generated earlier, which might throw off the program's results.

Which hash function shall we use with our counter mode parallel PRNG? The answer is to use a hash function that yields an output sequence that is statistically indistinguishable from a true random sequence, such as the numbers obtained by flipping coins or rolling dice. Most of the theory about PRNGs pertains to iterated mode PRNGs; for counter mode PRNGs there is little theory to guide us, and we must instead rely on empirical testing. As one simple example of a statistical test, consider using a PRNG to simulate a fair coin toss. If a random number's high-order bit is 0, the toss is heads; if it is 1, the toss is tails. Suppose we generate one million random numbers and observe 499,735 heads and 500,265 tails. That seems consistent with what we'd expect if the sequence were truly random. (We'd never expect to see precisely 500,000 heads and 500,000 tails, even when flipping a real coin.) But suppose we got 745,127 heads and 254,873 tails. That is almost certainly not random. A statistical test called the "chi-square test" can distinguish these cases; the former case would pass the chi-square test and the latter case would fail. In Chapter 31, we will examine another statistical test, the "Kolmogorov-Smirnov test."

There are test suites that apply a battery of statistical tests to a PRNG and report which tests failed. Passing all the tests increases our confidence in the PRNG as a source of random numbers. George Marsaglia's **Diehard** test suite, first published in 1995, has become the de facto standard for testing PRNGs. Pierre L'Ecuyer's and Richard Simard's **Crush** test suite, first published in 2002 as part of their TestU01 PRNG software library, provides more tests and more stringent tests than Diehard. A PRNG that passes Diehard might not pass Crush.

We want a counter-mode PRNG with a hash function that passes Crush. William Press *et al.* recommend the following 64-bit composite hash function, which does indeed pass both Diehard and Crush. The LCG and MCG parameters were found by Pierre L'Ecuyer and his colleagues to give hash functions with good randomness properties.

$$\begin{aligned}
H(x) &= H_4(H_3(H_2(H_1(x)))) \\
H_1(x) &= LCG(x; a, b, 2^{64}) \\
&\qquad a = 3935559000370003845 \\
&\qquad b = 2691343689449507681 \\
H_2(x) &= XorShiftRight(x; 21, 37, 4) \\
H_3(x) &= MCG(x; c, 2^{64}) \\
&\qquad c = 4768777513237032717 \\
H_4(x) &= XorShiftLeft(x; 20, 41, 5)
\end{aligned} \tag{14.9}$$

While a PRNG algorithm generates random numbers that are integers, often we want random *floating-point* numbers. One way to get random floating-point numbers is to divide the PRNG's output value by the largest possible output value (using floating-point division, not integer division). For a 64-bit PRNG, divide the output by $2^{64}$. This yields a random number in the range 0.0 to 1.0.

## 14.3 A Parallel PRNG Class

Taking the foregoing considerations into account, we can now design a PRNG suitable for random number generation in parallel programs. This is class edu.rit.util.Random in the Parallel Java Library.

**Algorithm**. Class edu.rit.util.Random uses the 64-bit hash function (14.9) in counter mode, which passes Diehard and Crush.

**Period**. With a 64-bit counter, class edu.rit.util.Random's period is $2^{64}$, or about twenty billion billion ($2 \times 10^{19}$).

**Types of random numbers**. Class edu.rit.util.Random has the methods `nextBoolean()`, `nextInt()`, `nextFloat()`, and `nextDouble()` to generate the next random number as type `boolean`, `int`, `float`, or `double`, respectively.

**Skipping**. To support the leapfrogging and sequence splitting techniques, class edu.rit.util.Random has a `skip()` method to skip over the next *n* random numbers in the sequence. Class edu.rit.util.Random also has variations of the `nextBoolean()`, `nextInt()`, `nextFloat()`, and `nextDouble()` methods to skip over the next *n*–1 random numbers in the sequence and generate the *n*-th random number.

**Not multiple-thread safe**. Unlike class java.util.Random, class edu.rit.util.Random is *not* multiple-thread safe. In parallel programs, PRNGs are usually per-thread variables for performance reasons, as in the PiSmp2 program in Chapter 13. But if only one thread ever calls methods on the PRNG, the PRNG class does not need to be multiple-thread safe, and omitting the thread synchronization code (atomic compare-and-set) reduces the parallel program's running time even further.

**Serializable**. An instance of class edu.rit.util.Random can serialize its state into an output stream, a file, for example. This can be useful for checkpointing the state of a parallel computation into a file so the program can stop temporarily and later pick up where it left off.

**Usage**. To create a PRNG object, call a static factory method in class edu.rit.util.Random, passing the seed as an argument.

```
Random prng = Random.getInstance (seed);
```

**Alternate algorithms**. The `getInstance()` factory method looks at the `"pj.prng"` Java system property to determine which PRNG algorithm to use. If this property is not specified, the `getInstance()` method creates an instance of the default PRNG described earlier. If the `"pj.prng"` property is specified, it gives the name of a class to instantiate. For example, the command:

```
$ java -Dpj.prng=SpecialRandom . . .
```

causes the `getInstance()` method to create an instance of class SpecialRandom (which must be a subclass of class edu.rit.util.Random). In this way, you can write your own PRNG classes and experiment with different PRNG algorithms from the command line without rewriting and recompiling your parallel program.

## 14.4 Parallel Program with Sequence Splitting

Class PiSeq3 in the Parallel Java Library is a sequential version of the Monte Carlo $\pi$ program that uses class edu.rit.util.Random to generate random numbers instead of class java.util.Random. Class PiSmp3 is an SMP parallel version that uses class edu.rit.util.Random with sequence splitting to generate the same random points as class PiSeq3. Here is the source code for class PiSmp3.

```
package edu.rit.smp.monte;
import edu.rit.pj.LongForLoop;
import edu.rit.pj.ParallelTeam;
import edu.rit.pj.ParallelRegion;
import edu.rit.pj.reduction.SharedLong;
import edu.rit.util.Random;
public class PiSmp3
   {
   // Command line arguments.
   static long seed;
   static long N;

   // Number of points within the unit circle.
   static SharedLong count;

   /**
    * Main program.
    */
   public static void main
      (String[] args)
      throws Exception
      {
      // Start timing.
      long time = -System.currentTimeMillis();

      // Validate command line arguments.
      if (args.length != 2) usage();
      seed = Long.parseLong (args[0]);
      N = Long.parseLong (args[1]);

      // Generate n random points in the unit square, count how
      // many are in the unit circle.
      count = new SharedLong (0);
      new ParallelTeam().execute (new ParallelRegion()
         {
         public void run() throws Exception
            {
```

```
execute (0, N-1, new LongForLoop())
   {
   // Set up per-thread PRNG and counter.
```

The first of only two differences between class PiSmp3 and class PiSmp2 is the following line. We create an instance of class edu.rit.util.Random instead of class java.util.Random.

```
   Random prng_thread = Random.getInstance (seed);
   long count_thread = 0L;

   // Extra padding to avert cache interference.
   long p0, p1, p2, p3, p4, p5, p6, p7;
   long p8, p9, pa, pb, pc, pd, pe, pf;

   // Parallel loop body.
   public void run (long first, long last)
      {
```

The other difference is that each thread, after initializing its per-thread PRNG with the seed specified on the command line, skips its PRNG ahead. The number of positions to skip is given by the lower bound of the loop index range (`first`), multiplied by 2 because each loop iteration generates two random numbers. This implements the sequence splitting. Note that the thread must re-seed and re-skip its PRNG each time the thread calls the `run()` method; this is necessary in case the parallel loop schedule makes the thread execute more than one chunk of loop indexes.

```
      // Skip PRNG ahead to index <first>.
      prng_thread.setSeed (seed);
      prng_thread.skip (2 * first);

      // Generate random points.
      for (long i = first; i <= last; ++ i)
         {
         double x = prng_thread.nextDouble();
         double y = prng_thread.nextDouble();
         if (x*x + y*y <= 1.0) ++ count_thread;
         }
      }

   public void finish()
      {
      // Reduce per-thread counts into shared count.
      count.addAndGet (count_thread);
      }
```

```
            });
         }
      });

   // Stop timing.
   time += System.currentTimeMillis();

   // Print results.
   System.out.println
      ("pi = 4 * " + count + " / " + N + " = " +
       (4.0 * count.doubleValue() / N));
   System.out.println (time + " msec");
   }
}
```

Let's consider whether this new program has fixed the design flaws in the original Monte Carlo $\pi$ program from Chapter 13 (PiSmp). Table 14.1 (at the end of the chapter) gives the running-time measurements in milliseconds for the Monte Carlo $\pi$ program with parallel reduction and sequence splitting for various problem sizes $N$ from 1 billion to 50 billion darts, as well as the speedups, efficiencies, and *EDSFs* calculated from the running times. Comparing the PiSeq and PiSmp2 programs' running times from Chapter 13 to the PiSeq3 and PiSmp3 programs' running times shows that the latter programs are about four times faster. This improvement is due to switching the PRNG class.

Figure 14.8 plots the running-time metrics versus the number of processors. The speedups and efficiencies are better than in Chapter 13—almost perfect, in fact. The *EDSFs* are mostly below two thousandths, also better than in Chapter 13. This is due to eliminating the thread synchronization code in the PRNG class. As was the case for the PiSmp2 program, the parallel reduction pattern has eliminated the first design flaw—the slowdown as $K$ increases.

What about the second design flaw, that the parallel program gives different answers when run on different numbers of processors? With a seed of 142857 and with $N = 1$ billion darts, the PiSeq3 program printed 3.141581704 as the estimate for $\pi$. The PiSmp3 program printed the same answer for $K = 1$ to 8 processors. The same is true for the other problem sizes. The sequence splitting technique has eliminated the second design flaw.

Here are the estimates for $\pi$ the PiSmp3 program calculated with a seed of 142857 for various numbers of iterations $N$, as well as the relative errors in the estimates:

| *N* | *Estimate* | *Relative Error* |
|---:|---:|---:|
| 1G | 3.14158170400 | $3.49\times10^{-6}$ |
| 2G | 3.14156025000 | $1.03\times10^{-5}$ |
| 5G | 3.14157178880 | $6.64\times10^{-6}$ |
| 10G | 3.14157868680 | $4.45\times10^{-6}$ |
| 20G | 3.14160519400 | $3.99\times10^{-6}$ |
| 50G | 3.14159556112 | $9.25\times10^{-7}$ |

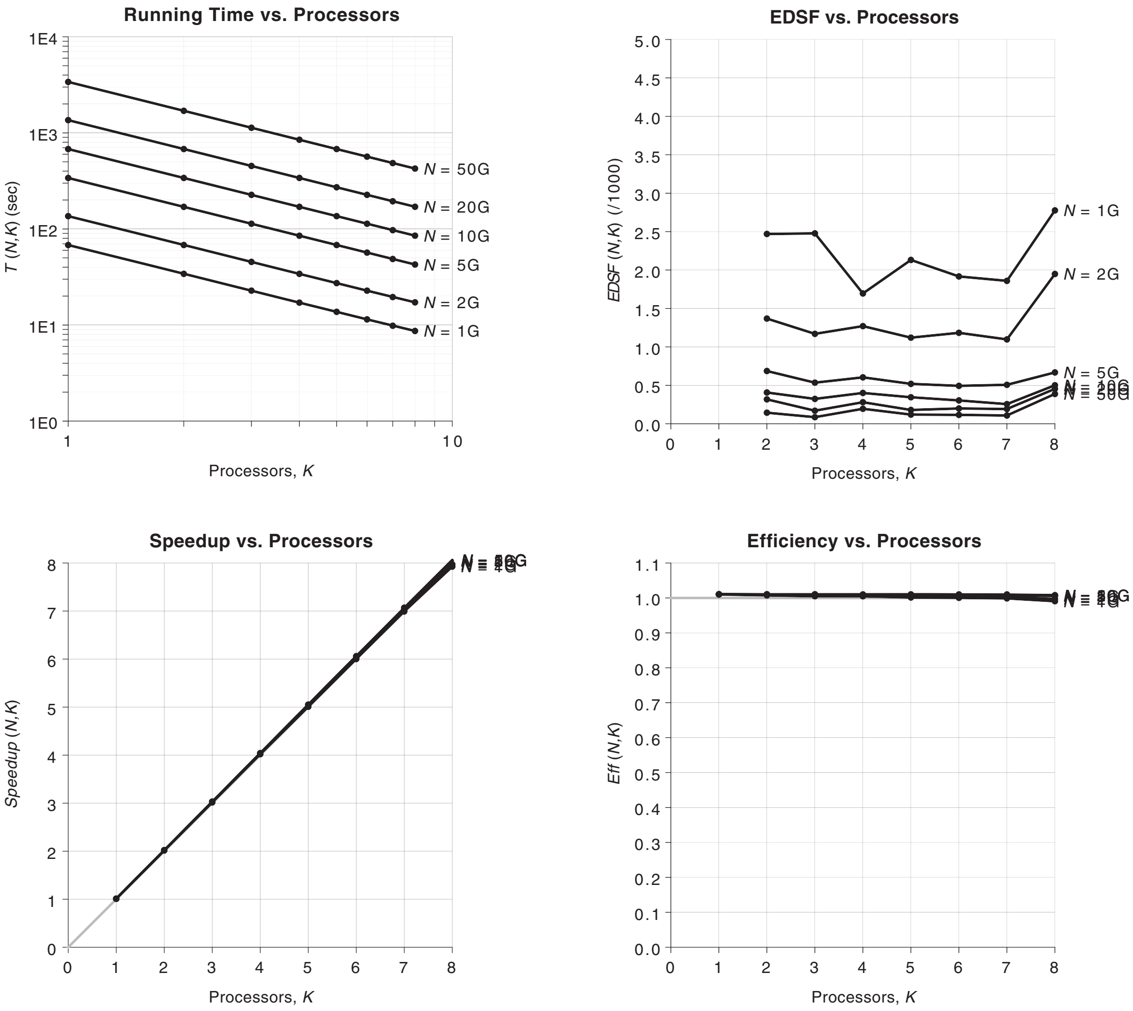


**Figure 14.8** PiSeq3/PiSmp3 running time metrics

## 14.5 Parting Remarks

Designing a parallel Monte Carlo program to compute results identical to its sequential counterpart requires extra code to implement leapfrogging or sequence splitting and requires using a special parallel PRNG with a skip operation. But why go to this extra effort? After all, because we are using random numbers to solve the problem, we don't expect the program to give the precise answer; we only expect an answer that is statistically close. If the parallel program's answer is just as close to the true answer as the sequential program's answer, statistically speaking, isn't that good enough?

No, that is not good enough. Unfortunately, a Monte Carlo program's use of random numbers can mask the existence of bugs. If the parallel program's output is not the same as the sequential program's, we don't know whether it is due to the random sequences or to a bug in the program. We saw this with the Monte Carlo $\pi$ programs in Chapter 13. The parallel program's outputs for various numbers of processors, although different from each other, were all within 0.01% of the true value of $\pi$. If we had been less suspicious, we might have attributed that to random variation and we might not have discovered the bug, that all the threads were generating the same sequences of random numbers.

On the other hand, if the program is designed so that the parallel version is supposed to produce output identical to the sequential version no matter how many processors are used, and the parallel version's output is not identical to the sequential version's, then there is definitely a bug somewhere. We still won't be able to detect a bug that affects the sequential and parallel versions the same way. But because sequential programs are easier to debug than parallel programs, we ought to be able to detect those kinds of bugs when developing the sequential version.

## 14.6 For Further Information

On PRNG algorithms and statistical tests of randomness:

- P. L'Ecuyer. Random numbers for simulation. *Communications of the ACM*, 33(10):85–97, October 1990.
- P. L'Ecuyer. Random number generation. In J. Banks, editor. *Handbook of Simulation: Principles, Methodology, Advances, Applications, and Practice*. Wiley, 1998, Chapter 4.
- D. Knuth. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms, Third Edition*. Addison-Wesley, 1998, Chapter 3.
- W. Press, S. Teukolsky, W. Vetterling, and B. Flannery. *Numerical Recipes: The Art of Scientific Computing, Third Edition*. Cambridge University Press, 2008, Chapter 7.

On good parameter values for MCGs and LCGs:

- P. L'Ecuyer, F. Blouin, and R. Couture. A search for good multiple recursive random number generators. *ACM Transactions on Modeling and Computer Simulation*, 3(2):87–98, April 1993.
- P. L'Ecuyer. Tables of linear congruential generators of different sizes and good lattice structure. *Mathematics of Computation*, 68(225):249–260, 1999.

On the xorshift generator:

- G. Marsaglia. Xorshift RNGs. *Journal of Statistical Software*, 8(14):1–6, July 2003.

On the Diehard PRNG test suite:

- G. Marsaglia. Diehard Battery of Tests of Randomness v0.2 beta. http://www.cs.hku.hk/~diehard/

On the Crush PRNG test suite and the TestU01 PRNG software library:

- P. L'Ecuyer and R. Simard. TestU01: a C library for empirical testing of random number generators. *ACM Transactions on Mathematical Software*, 33(4):22, August 2007.

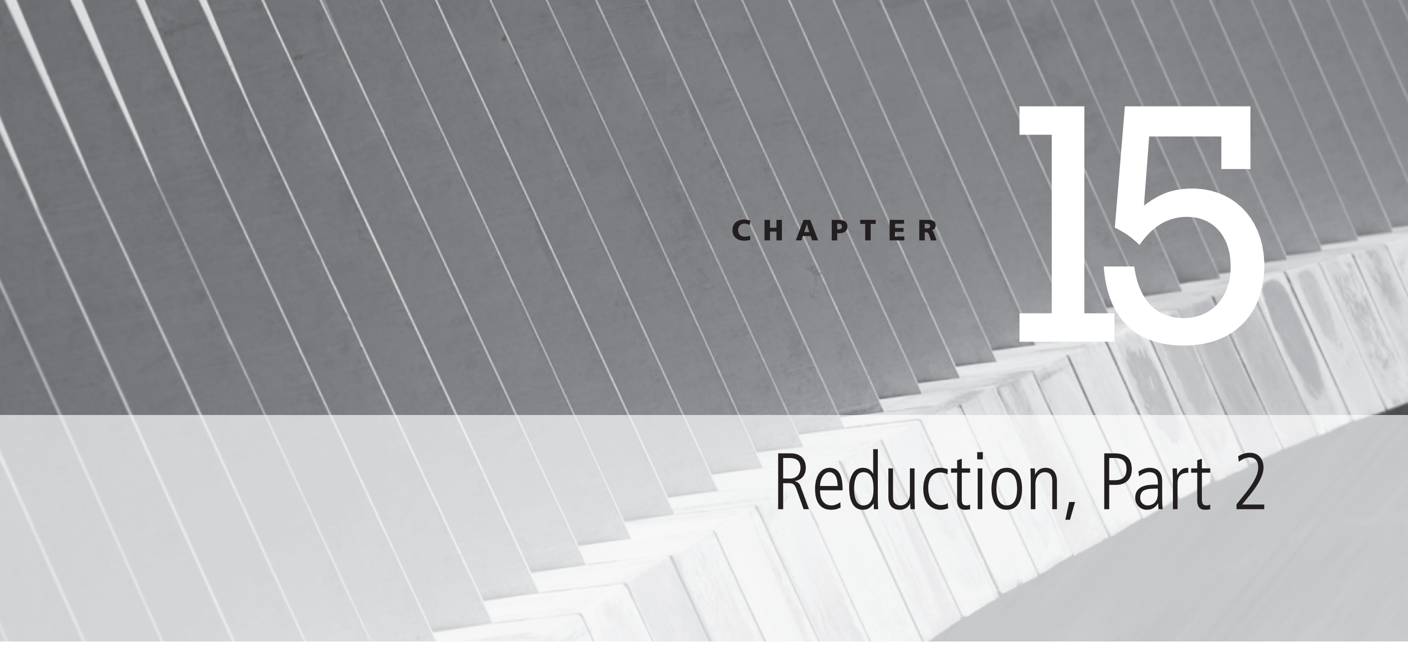
- R. Simard. TestU01: Empirical testing of random number generators. http://www.iro.umontreal.ca/~simardr/testu01/tu01.html

On parallel PRNG design in Java:

- P. Coddington and A. Newell. JAPARA—a Java random number generator library for high-performance computing. In *Proceedings of the 18th IEEE International Parallel and Distributed Processing Symposium (IPDPS'04)*, April 2004, page 156.

**Table 14.1** PiSeq3/PiSmp3 running-time metrics

| *N* | *K* | *T* | *Spdup* | *Eff* | *EDSF* | *N* | *K* | *T* | *Spdup* | *Eff* | *EDSF* |
|---:|---:|---:|---:|---:|---:|---:|---:|---:|---:|---:|---:|
| 1G | seq | 68702 | | | | 10G | seq | 685992 | | | |
| 1G | 1 | 68030 | 1.010 | 1.010 | | 10G | 1 | 678884 | 1.010 | 1.010 | |
| 1G | 2 | 34099 | 2.015 | 1.007 | 0.002 | 10G | 2 | 339580 | 2.020 | 1.010 | 0.000 |
| 1G | 3 | 22789 | 3.015 | 1.005 | 0.002 | 10G | 3 | 226441 | 3.029 | 1.010 | 0.000 |
| 1G | 4 | 17094 | 4.019 | 1.005 | 0.002 | 10G | 4 | 169925 | 4.037 | 1.009 | 0.000 |
| 1G | 5 | 13722 | 5.007 | 1.001 | 0.002 | 10G | 5 | 135964 | 5.045 | 1.009 | 0.000 |
| 1G | 6 | 11447 | 6.002 | 1.000 | 0.002 | 10G | 6 | 113319 | 6.054 | 1.009 | 0.000 |
| 1G | 7 | 9827 | 6.991 | 0.999 | 0.002 | 10G | 7 | 97132 | 7.062 | 1.009 | 0.000 |
| 1G | 8 | 8669 | 7.925 | 0.991 | 0.003 | 10G | 8 | 85158 | 8.056 | 1.007 | 0.001 |
| 2G | seq | 137300 | | | | 20G | seq | 1371888 | | | |
| 2G | 1 | 135906 | 1.010 | 1.010 | | 20G | 1 | 1357531 | 1.011 | 1.011 | |
| 2G | 2 | 68046 | 2.018 | 1.009 | 0.001 | 20G | 2 | 678981 | 2.021 | 1.010 | 0.000 |
| 2G | 3 | 45408 | 3.024 | 1.008 | 0.001 | 20G | 3 | 452664 | 3.031 | 1.010 | 0.000 |
| 2G | 4 | 34106 | 4.026 | 1.006 | 0.001 | 20G | 4 | 339668 | 4.039 | 1.010 | 0.000 |
| 2G | 5 | 27303 | 5.029 | 1.006 | 0.001 | 20G | 5 | 271701 | 5.049 | 1.010 | 0.000 |
| 2G | 6 | 22785 | 6.026 | 1.004 | 0.001 | 20G | 6 | 226480 | 6.057 | 1.010 | 0.000 |
| 2G | 7 | 19543 | 7.026 | 1.004 | 0.001 | 20G | 7 | 194156 | 7.066 | 1.009 | 0.000 |
| 2G | 8 | 17220 | 7.973 | 0.997 | 0.002 | 20G | 8 | 170231 | 8.059 | 1.007 | 0.000 |
| 5G | seq | 343064 | | | | 50G | seq | 3429530 | | | |
| 5G | 1 | 339525 | 1.010 | 1.010 | | 50G | 1 | 3393498 | 1.011 | 1.011 | |
| 5G | 2 | 169879 | 2.019 | 1.010 | 0.001 | 50G | 2 | 1696993 | 2.021 | 1.010 | 0.000 |
| 5G | 3 | 113296 | 3.028 | 1.009 | 0.001 | 50G | 3 | 1131360 | 3.031 | 1.010 | 0.000 |
| 5G | 4 | 85035 | 4.034 | 1.009 | 0.001 | 50G | 4 | 848870 | 4.040 | 1.010 | 0.000 |
| 5G | 5 | 68046 | 5.042 | 1.008 | 0.001 | 50G | 5 | 679023 | 5.051 | 1.010 | 0.000 |
| 5G | 6 | 56727 | 6.048 | 1.008 | 0.000 | 50G | 6 | 565910 | 6.060 | 1.010 | 0.000 |
| 5G | 7 | 48651 | 7.052 | 1.007 | 0.001 | 50G | 7 | 485098 | 7.070 | 1.010 | 0.000 |
| 5G | 8 | 42639 | 8.046 | 1.006 | 0.001 | 50G | 8 | 425339 | 8.063 | 1.008 | 0.000 |



# CHAPTER 15

# Reduction, Part 2

in which we learn different ways to implement the parallel reduction pattern in an SMP parallel program; and we discover how each technique affects the parallel program's performance

# CHAPTER 15 Reduction, Part 2

## 15.1 Histogram of the Mandelbrot Set

Let's return to the problem of computing the Mandelbrot Set from Chapter 11. However, this time we want to calculate, not the image itself, but rather the *number of pixels* whose iteration count was 0, the number of pixels whose iteration count was 1, and so on, up to the maximum iteration count. This data is called the Mandelbrot Set image's **histogram**.

The program for computing a Mandelbrot Set histogram that we are about to examine takes these command-line arguments:

- *width*—Image width in pixels.
- *height*—Image height in pixels.
- *xcenter*—X coordinate of the image's center point.
- *ycenter*—Y coordinate of the image's center point.
- *resolution*—Image resolution in pixels per unit.
- *maxiter*—Maximum number of iterations for deciding whether a point is in the set.
- *outfile*—Output file name.

The first six command-line arguments are the same as the Mandelbrot Set programs in Chapter 11. The Mandelbrot Set histogram program prints the histogram data into the output file. Here is an example of the program's output, for a 3,200×3,200-pixel image of the whole Mandelbrot Set.

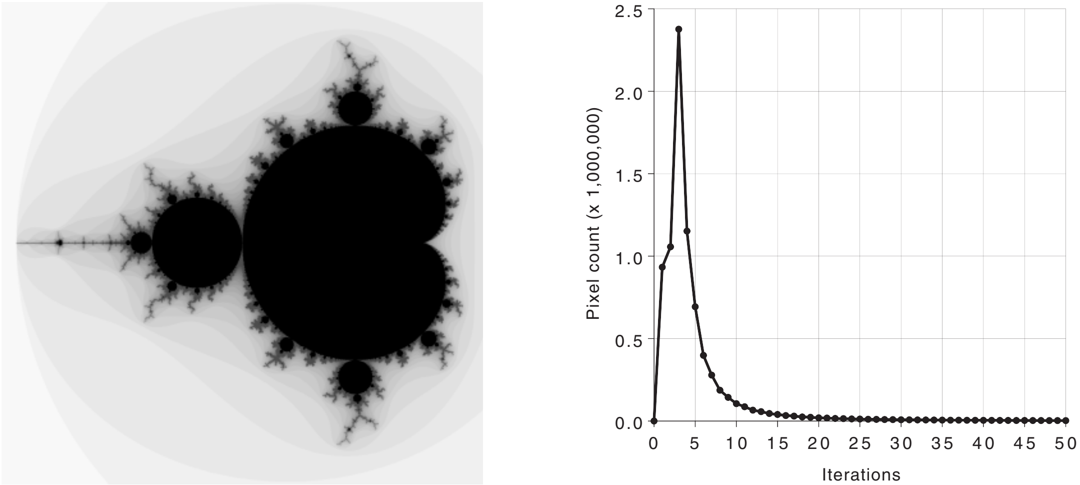


**Figure 15.1** Histogram of the Mandelbrot Set

```
$ java edu.rit.smp.fractal.MSHistogramSeq 3200 3200 -0.75 0 \
  1200 1000 out.txt; cat out.txt
0       0
1       933024
2       1056585
3       2376460
4       1151978
5       692974
. . .
995     6
996     2
997     0
998     6
999     14
1000    2175587
```

That is, there were no pixels with an iteration count of 0, there were 933,024 pixels with an iteration count of 1, there were 1,056,585 pixels with an iteration count of 2, and so on. Figure 15.1 shows the image and plots the histogram data (for iteration counts from 0 to 50) for the preceding command-line arguments.

Figure 15.2 shows the image and plots the histogram data (for iteration counts from 0 to 50) for a different set of parameters:

```
$ java edu.rit.smp.fractal.MSHistogramSeq 3200 3200 -0.55 0.6 \
  76800 1000 out.txt
```

The histogram plot shows that in this region of the Mandelbrot Set, the pixels tend to have higher iteration counts than in Figure 15.1.

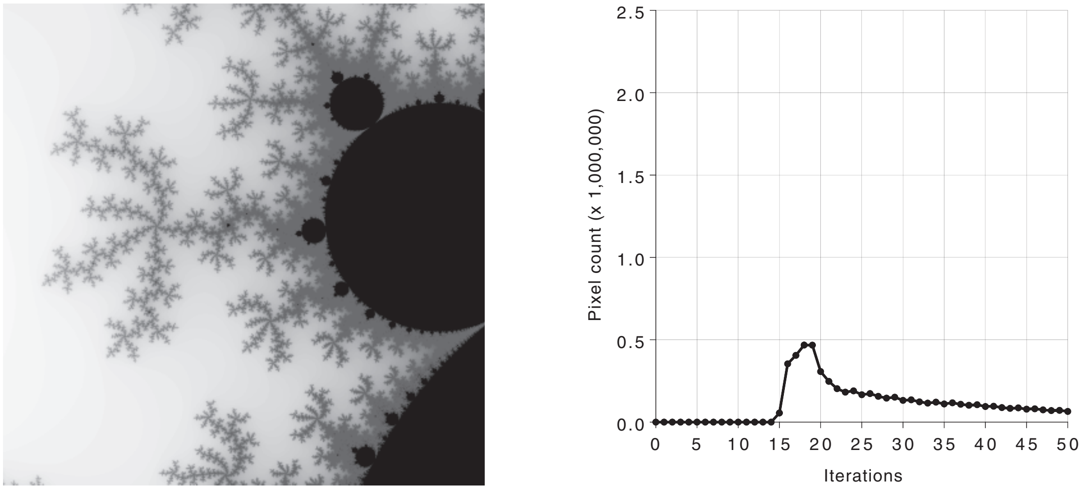


**Figure 15.2** Histogram of a portion of the Mandelbrot Set

## 15.2 Sequential Version

Here is the source code for class MSHistogramSeq, the sequential version of the Mandelbrot Set histogram program.

```
package edu.rit.smp.fractal;
import java.io.BufferedWriter;
import java.io.File;
import java.io.FileWriter;
import java.io.PrintWriter;
public class MSHistogramSeq
   {
   // Command line arguments.
   static int width;
   static int height;
   static double xcenter;
   static double ycenter;
   static double resolution;
   static int maxiter;
   static File outfile;

   // Initial pixel offsets from center.
   static int xoffset;
   static int yoffset;
```

In this program, instead of setting up an image, we'll set up an array of counters. The array index is the iteration count, from 0 to the maximum number of iterations. The array element at index *i* is the number of pixels whose iteration count is equal to *i*.

```
// Histogram (array of counters indexed by pixel value).
static int[] histogram;

/**
 * Mandelbrot Set main program.
 */
public static void main
   (String[] args)
   throws Exception
   {
   // Start timing.
   long t1 = System.currentTimeMillis();

   // Validate command line arguments.
   if (args.length != 7) usage();
   width = Integer.parseInt (args[0]);
   height = Integer.parseInt (args[1]);
   xcenter = Double.parseDouble (args[2]);
   ycenter = Double.parseDouble (args[3]);
   resolution = Double.parseDouble (args[4]);
   maxiter = Integer.parseInt (args[5]);
   outfile = new File (args[6]);

   // Initial pixel offsets from center.
   xoffset = -(width - 1) / 2;
   yoffset = (height - 1) / 2;

   // Create histogram.
   histogram = new int [maxiter + 1];

   long t2 = System.currentTimeMillis();

   // Compute all rows and columns.
   for (int r = 0; r < height; ++ r)
      {
      double y = ycenter + (yoffset - r) / resolution;
      for (int c = 0; c < width; ++ c)
         {
         double x = xcenter + (xoffset + c) / resolution;
         // Iterate until convergence.
         int i = 0;
         double aold = 0.0;
         double bold = 0.0;
         double a = 0.0;
         double b = 0.0;
         double zmagsqr = 0.0;
         while (i < maxiter && zmagsqr <= 4.0)
            {
            ++ i;
            a = aold*aold - bold*bold + x;
            b = 2.0*aold*bold + y;
            zmagsqr = a*a + b*b;
            aold = a;
            bold = b;
            }
```

Instead of setting the pixel's color in the image, we increase the counter corresponding to the pixel's iteration count *i*.

```
      // Increment histogram counter for pixel value.
      ++ histogram[i];
      }
   }

long t3 = System.currentTimeMillis();

// Print histogram.
PrintWriter out =
   new PrintWriter
      (new BufferedWriter
         (new FileWriter (outfile)));
for (int i = 0; i <= maxiter; ++ i)
   {
   out.print (i);
   out.print ('\t');
   out.print (histogram[i]);
   out.println();
   }
out.close();

// Stop timing.
long t4 = System.currentTimeMillis();
System.out.println ((t2-t1) + " msec pre");
System.out.println ((t3-t2) + " msec calc");
```

```
        System.out.println ((t4-t3) + " msec post");
        System.out.println ((t4-t1) + " msec total");
        }
```

## 15.3 Parallel Version without Reduction

In Chapter 11's parallel program for computing an image of the Mandelbrot Set, the pixel RGB data was stored in a shared variable, an integer matrix (type `int[][]`), accessed by all the parallel team threads. However, in that program, no thread synchronization was necessary when accessing the shared matrix because each matrix *element* (pixel) was written by only one thread.

For the Mandelbrot Set histogram program, this is no longer the case. In general, each element of the shared histogram data array can be accessed by every parallel team thread. Because incrementing an array element is an update operation, the threads conflict with each other when accessing the array, and thread synchronization is required. We are about to look at three different ways to implement this thread synchronization as well as each implementation's effect on the program's performance.

The first version of the parallel program uses a multiple-thread-safe shared variable, an instance of class edu.rit.pj.reduction.SharedIntegerArray, to hold the histogram data (Figure 15.3). This class encapsulates an array of `ints` and provides several methods to manipulate the array, including a method to increment a given array element. Each parallel team thread calls this method on the shared variable to increment the proper histogram counter (array element) for each pixel. However, this first version does not use the parallel reduction pattern. Each thread increments the shared histogram counters directly.

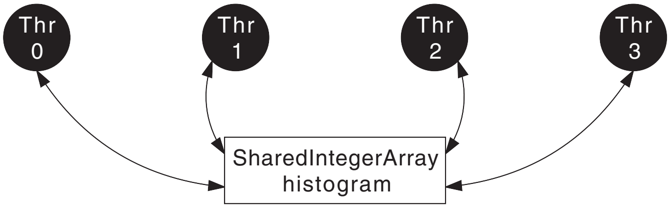


**Figure 15.3** Threads accessing shared variable, without reduction

Here is the source code for class MSHistogramSmp, the first parallel version.

```
package edu.rit.smp.fractal;
import edu.rit.pj.IntegerForLoop;
import edu.rit.pj.ParallelRegion;
import edu.rit.pj.ParallelTeam;
import edu.rit.pj.reduction.SharedIntegerArray;
import java.io.BufferedWriter;
import java.io.File;
import java.io.FileWriter;
import java.io.PrintWriter;
public class MSHistogramSmp
```

```
{
// Command line arguments.
static int width;
static int height;
static double xcenter;
static double ycenter;
static double resolution;
static int maxiter;
static File outfile;

// Initial pixel offsets from center.
static int xoffset;
static int yoffset;
```

We declare a SharedIntegerArray variable to hold the histogram data.

```
// Histogram (array of counters indexed by pixel value).
static SharedIntegerArray histogram;

/**
 * Mandelbrot Set main program.
 */
public static void main
   (String[] args)
   throws Exception
   {
   // Start timing.
   long t1 = System.currentTimeMillis();

   // Validate command line arguments.
   if (args.length != 7) usage();
   width = Integer.parseInt (args[0]);
   height = Integer.parseInt (args[1]);
   xcenter = Double.parseDouble (args[2]);
   ycenter = Double.parseDouble (args[3]);
   resolution = Double.parseDouble (args[4]);
   maxiter = Integer.parseInt (args[5]);
   outfile = new File (args[6]);

   // Initial pixel offsets from center.
   xoffset = -(width - 1) / 2;
   yoffset = (height - 1) / 2;
```

We initialize the histogram array with a length of `maxiter` (the maximum iteration count) plus 1, yielding array indexes of 0 through `maxiter`. The SharedIntegerArray constructor initializes each array element to 0.

```
// Create histogram.
histogram = new SharedIntegerArray (maxiter + 1);

long t2 = System.currentTimeMillis();
```

As we did with the Mandelbrot Set image program in Chapter 11, we change the sequential version's outer loop over the pixel rows to be a parallel loop inside a parallel region, executed by a parallel team of threads. The inner loop over the pixel columns remains a regular loop. Because each loop iteration does a different amount of work, we must balance the load by specifying a dynamic or guided schedule when we run the program.

```
// Parallel computation region.
new ParallelTeam().execute (new ParallelRegion()
   {
   public void run() throws Exception
      {
      execute (0, height-1, new IntegerForLoop()
         {
         public void run (int first, int last)
            {
            // Compute all rows and columns.
            for (int r = first; r <= last; ++ r)
               {
               double y =
                  ycenter + (yoffset - r) / resolution;

               for (int c = 0; c < width; ++ c)
                  {
                  double x =
                     xcenter + (xoffset + c) / resolution;

                  // Iterate until convergence.
                  int i = 0;
                  double aold = 0.0;
                  double bold = 0.0;
                  double a = 0.0;
                  double b = 0.0;
                  double zmagsqr = 0.0;
                  while (i < maxiter && zmagsqr <= 4.0)
                     {
                     ++ i;
                     a = aold*aold - bold*bold + x;
                     b = 2.0*aold*bold + y;
                     zmagsqr = a*a + b*b;
```

```
                        aold = a;
                        bold = b;
                        }
```

Each thread increments the proper histogram counter directly using the multiple-thread-safe `incrementAndGet()` method.

```
                     // Increment histogram counter.
                     histogram.incrementAndGet (i);
                     }
                  }
               }
            });
         }
      });

   long t3 = System.currentTimeMillis();

   // Print histogram.
   PrintWriter out =
      new PrintWriter
         (new BufferedWriter
            (new FileWriter (outfile)));
   for (int i = 0; i <= maxiter; ++ i)
      {
      out.print (i);
      out.print ('\t');
      out.print (histogram.get (i));
      out.println();
      }
   out.close();

   // Stop timing.
   long t4 = System.currentTimeMillis();
   System.out.println ((t2-t1) + " msec pre");
   System.out.println ((t3-t2) + " msec calc");
   System.out.println ((t4-t3) + " msec post");
   System.out.println ((t4-t1) + " msec total");
   }
}
```

We'll put off measuring the MSHistogramSmp program's performance until we can compare it to the next version of the program.

## 15.4 Reduction Operators

As we saw in Chapter 13, a parallel program might perform poorly if multiple threads have to synchronize with each other repeatedly in a parallel loop. Using the parallel reduction pattern restores good performance.

The elements of the parallel reduction pattern are threefold:

1. The program has a global shared reduction variable or variables to hold the complete result computed by the whole program.
2. Each parallel team thread has a per-thread variable or variables that hold the partial result computed by that thread.
3. A computation is executed to combine the threads' partial results, storing the complete result in the global shared reduction variable or variables.

While the reduction step (Step 3) could be any computation, often the reduction consists of combining the partial results with a **reduction operator**, a binary operation (Figure 15.4). The reduction variable is initialized to a certain value. The contents of the reduction variable and the contents of one per-thread variable are fed into the reduction operator, the reduction operator computes a result from those two inputs, and the result is stored back into the shared reduction variable. Repeating this process for each per-thread variable yields the program's complete result, which is stored in the reduction variable. For example, if the reduction variable is initialized to 0 and the reduction operator is addition, the reduction variable ends up holding the sum of all the per-thread variables.

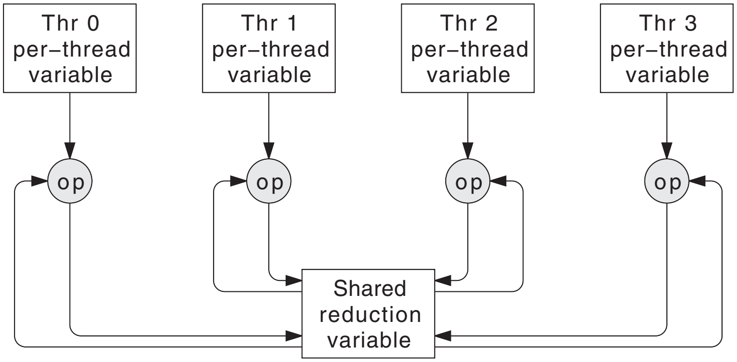

Thr 0 per–thread variable
Thr 1 per–thread variable
Thr 2 per–thread variable
Thr 3 per–thread variable
op
op
op
op
Shared reduction variable

**Figure 15.4** Reduction operator *op*

However, the program has no control over when each thread performs the reduction. The threads' partial results could be accumulated into the reduction variable in any order. Thus, the reduction operator must yield the same final answer regardless of the order in which the partial results are reduced. That is, the binary operation must be commutative and associative.

Parallel Java provides class edu.rit.pj.reduction.IntegerOp, an abstraction of a reduction operator with integer operands. The class defines one method that computes (`x` *op* `y`) for some binary operation *op*.

```
   public abstract int op (int x, int y);
```

Class IntegerOp provides objects for four common reduction operators: `SUM`, `PRODUCT`, `MINIMUM`, and `MAXIMUM`. As an example, here is how the Parallel Java Library implements class IntegerOp's `SUM` operator. It is just an object whose `op()` method returns the sum of its arguments. (Shortly we will see a program that uses the `SUM` operator to do a reduction.)

```
   public static final IntegerOp SUM = new IntegerOp()
      {
      public int op (int x, int y)
         {
         return x + y;
         }
      };
```

You can also define your own reduction operator classes. If, say, your program's complete result must be the sum of the partial results modulo some fixed number *M*, you can define the following reduction operator class:

```
import edu.rit.pj.reduction.IntegerOp;
public class ModularSum
   extends IntegerOp
   {
   private int M;

   public ModularSum (int M)
      {
      this.M = M;
      }

   public int op (int x, int y)
      {
      return (x + y) % M;
      }
   }
```

Besides class IntegerOp, Parallel Java has reduction operator classes for all the primitive types as well as nonprimitive types (objects).

Each shared reduction variable class in package edu.rit.pj.reduction has a method to perform a reduction using a reduction operator. For example, class SharedInteger has the following method:

```
   public int reduce (int value, IntegerOp op);
```

Calling `x.reduce (y, op)`—where `x` is a reduction variable of type SharedInteger, `y` is a per-thread variable of type `int`, and `op` is a reduction operator—replaces the contents of `x` with (`x` *op* `y`) and returns the new contents of `x`. The `reduce()` method uses an atomic compare-and-set operation for thread synchronization (it is multiple-thread safe).

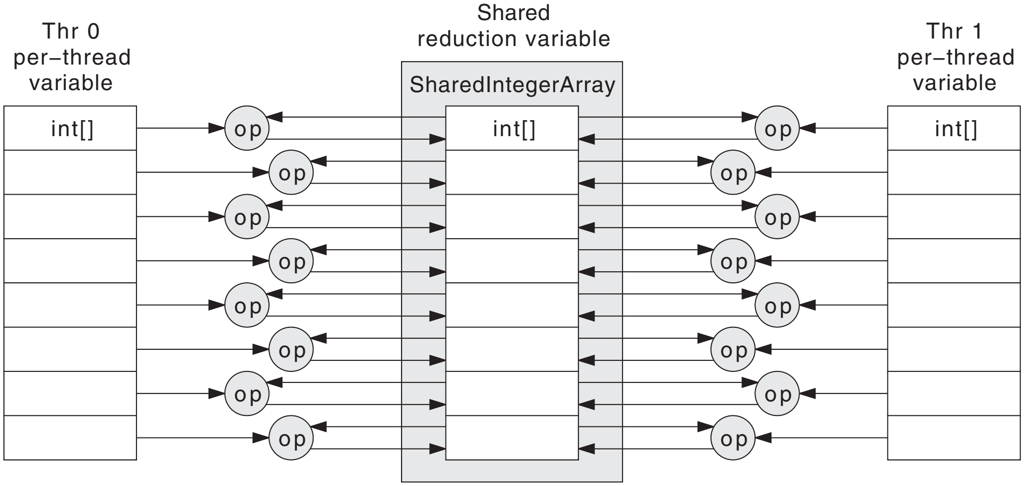


**Figure 15.5** Reduction operator *op* acting on an array

Likewise, each shared reduction *array* class in package edu.rit.pj.reduction has a method to perform a reduction using a reduction operator (Figure 15.5). For example, class SharedIntegerArray has the following method:

```
    public void reduce (int[] value, IntegerOp op);
```

Calling `x.reduce (y, op)`—where `x` is a reduction variable of type SharedIntegerArray, `y` is a per-thread variable of type `int[]`, and `op` is a reduction operator—replaces the contents of `x[i]` with (`x[i]` *op* `y[i]`) for every index `i` in the array `x`. Again, the `reduce()` method is multiple-thread safe; each element of `x` is updated using an atomic compare-and-set operation.

## 15.5 Parallel Version with Reduction

The second version of the parallel Mandelbrot Set histogram program uses the reduction pattern (Figure 15.6). Each thread computes the histogram for a portion of the pixels and stores the histogram data in a per-thread variable. The per-thread histograms are combined using a reduction operator to form the complete histogram stored in a shared variable. Here is the source code for class MSHistogramSmp2.

```
package edu.rit.smp.fractal;
import edu.rit.pj.IntegerForLoop;
import edu.rit.pj.ParallelRegion;
import edu.rit.pj.ParallelTeam;
import edu.rit.pj.reduction.IntegerOp;
```

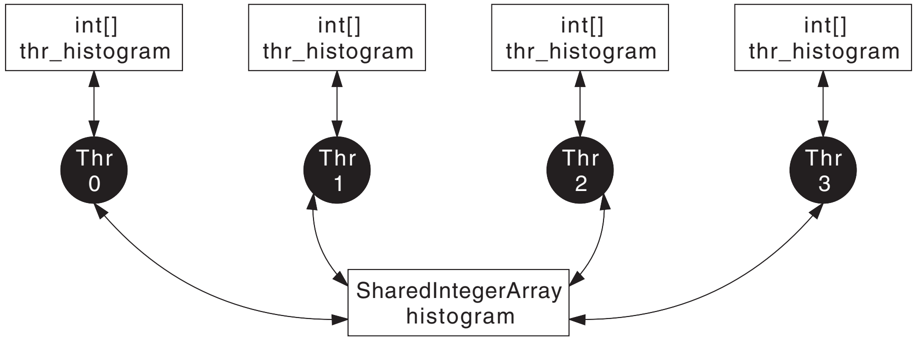


**Figure 15.6** Threads using reduction with a reduction operator

```
import edu.rit.pj.reduction.SharedIntegerArray;
import java.io.BufferedWriter;
import java.io.File;
import java.io.FileWriter;
import java.io.PrintWriter;
public class MSHistogramSmp2
   {
   // Command line arguments.
   static int width;
   static int height;
   static double xcenter;
   static double ycenter;
   static double resolution;
   static int maxiter;
   static File outfile;

   // Initial pixel offsets from center.
   static int xoffset;
   static int yoffset;
```

This program still has a shared histogram variable of type SharedIntegerArray, but now it is acting as a reduction variable.

```
   // Histogram (array of counters indexed by pixel value).
   static SharedIntegerArray histogram;

   /**
    * Mandelbrot Set main program.
    */
   public static void main
```

```
         (String[] args)
         throws Exception
         {
         // Start timing.
         long t1 = System.currentTimeMillis();

         // Validate command line arguments.
         if (args.length != 7) usage();
         width = Integer.parseInt (args[0]);
         height = Integer.parseInt (args[1]);
         xcenter = Double.parseDouble (args[2]);
         ycenter = Double.parseDouble (args[3]);
         resolution = Double.parseDouble (args[4]);
         maxiter = Integer.parseInt (args[5]);
         outfile = new File (args[7]);

         // Initial pixel offsets from center.
         xoffset = -(width - 1) / 2;
         yoffset = (height - 1) / 2;

         // Create histogram.
         histogram = new SharedIntegerArray (maxiter + 1);

         long t2 = System.currentTimeMillis();

         // Parallel computation region.
         new ParallelTeam().execute (new ParallelRegion()
            {
            public void run() throws Exception
               {
               execute (0, height-1, new IntegerForLoop()
                  {
```

Continuing the parallel reduction pattern, each parallel team thread also creates its own per-thread histogram variable. Because only one thread accesses this variable, it does not need to be multiple thread safe, and type `int[]` suffices. However, to avert cache interference between the threads, we add 128 padding bytes (32 `int`s, 4 bytes each) to the array's storage block, and we add 128 padding bytes (16 `long` fields, 8 bytes each) to the IntegerForLoop subclass's storage block.

```
                  // Per-thread histogram, plus extra padding.
                  int[] thr_histogram = new int [maxiter + 1 + 32];
                  long p0, p1, p2, p3, p4, p5, p6, p7;
                  long p8, p9, pa, pb, pc, pd, pe, pf;
```

```
public void run (int first, int last)
   {
   // Compute all rows and columns.
   for (int r = first; r <= last; ++ r)
      {
      double y =
         ycenter + (yoffset - r) / resolution;

      for (int c = 0; c < width; ++ c)
         {
         double x =
            xcenter + (xoffset + c) / resolution;

         // Iterate until convergence.
         int i = 0;
         double aold = 0.0;
         double bold = 0.0;
         double a = 0.0;
         double b = 0.0;
         double zmagsqr = 0.0;
         while (i < maxiter && zmagsqr <= 4.0)
            {
            ++ i;
            a = aold*aold - bold*bold + x;
            b = 2.0*aold*bold + y;
            zmagsqr = a*a + b*b;
            aold = a;
            bold = b;
            }
```

Inside the parallel for loop, each thread increments its own per-thread histogram counter.

```
         // Increment histogram counter.
         ++ thr_histogram[i];
         }
      }
   }
```

To finish the parallel reduction, each thread uses the `IntegerOp.SUM` reduction operator to add its own per-thread histogram into the shared reduction variable. Putting this code in the parallel for loop's `finish()` method causes each thread to perform the reduction after all the thread's loop iterations have finished.

```
// Reduce per-thread histogram into global
// histogram.
```

```
            public void finish()
               {
               histogram.reduce (thr_histogram, IntegerOp.SUM);
               }
            });
         }
      });
   long t3 = System.currentTimeMillis();
   // Print histogram.
   PrintWriter out =
      new PrintWriter
         (new BufferedWriter
            (new FileWriter (outfile)));
   for (int i = 0; i <= maxiter; ++ i)
      {
      System.out.print (i);
      System.out.print ('\t');
      System.out.print (histogram.get (i));
      System.out.println();
      }
   out.close();

   // Stop timing.
   long t4 = System.currentTimeMillis();
   System.err.println ((t2-t1) + " msec pre");
   System.err.println ((t3-t2) + " msec calc");
   System.err.println ((t4-t3) + " msec post");
   System.err.println ((t4-t1) + " msec total");
   }
}
```

## 15.6 Performance Comparison

Before going on to the third parallel version of the Mandelbrot Set histogram program, let's compare the performance of the first and second versions. Table 15.1 (at the end of the chapter) lists, and Figure 15.7 plots, the MSHistogramSmp program's running times, speedups, and efficiencies for the same input data as in Chapter 11, using a guided schedule for load balancing, without parallel reduction. Table 15.2 lists, and Figure 15.8 plots, the same for the MSHistogramSmp2 program, with parallel reduction.

The plots clearly show that the program with reduction performs better than the program without reduction. The running times for the program with reduction are less than for the program without reduction, ranging from 1 percent less on one processor to 20 percent less on eight processors. Furthermore, the speedups and efficiencies are higher for the program with reduction. This is because the reduction pattern eliminates almost all of the thread synchronization overhead incurred when accessing the shared histogram variable.

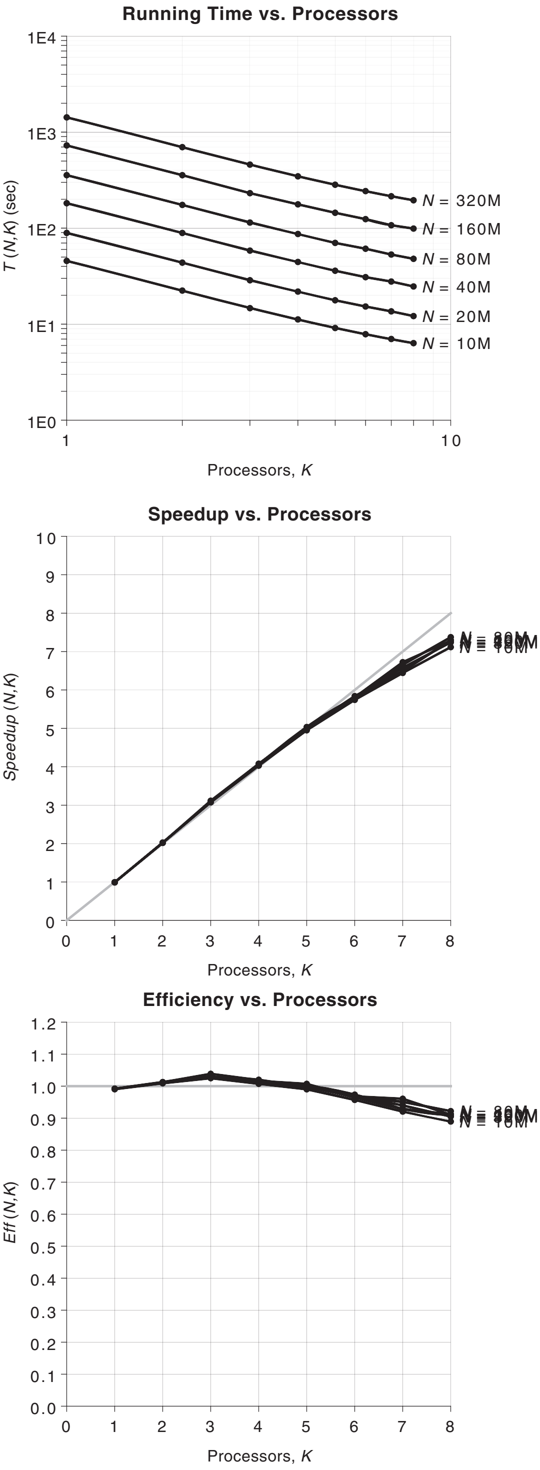


**Figure 15.7** MSHistogramSeq/MSHistogramSmp running-time metrics, without reduction

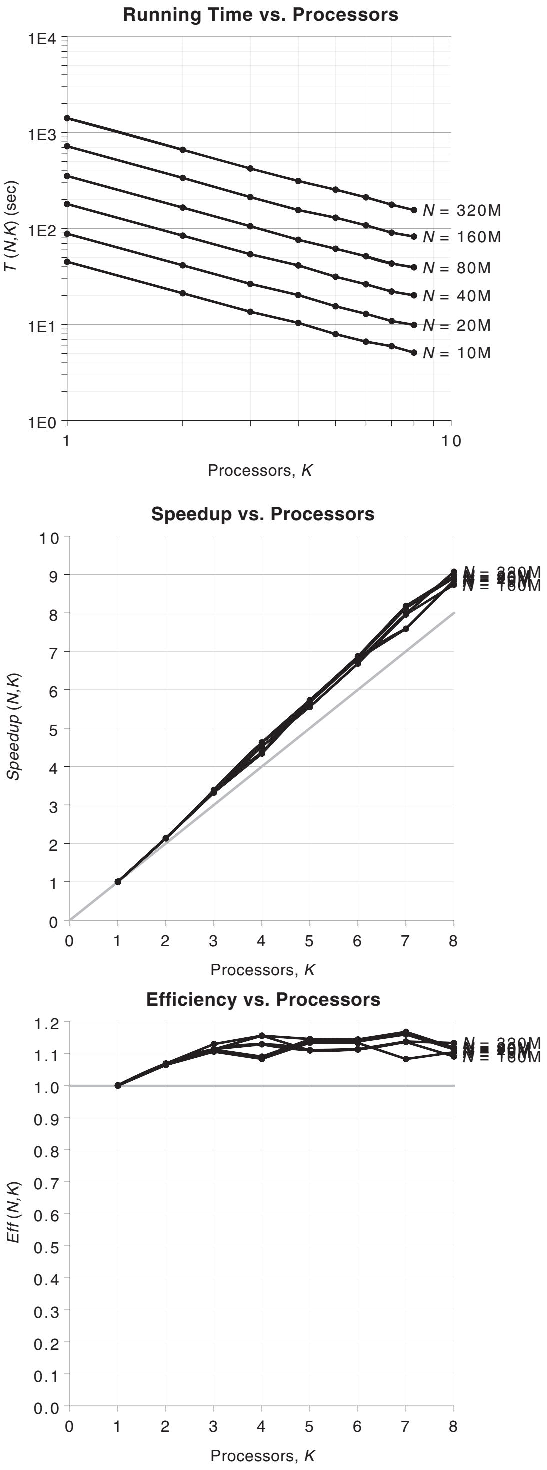


**Figure 15.8** MSHistogramSeq/MSHistogramSmp2 running-time metrics, with reduction

In fact, the program with reduction is showing *superlinear* speedups. This is due to the Java virtual machine's just-in-time compiler coupled with the parallel for loop's guided schedule—the *JIT compiler effect* we observed in Chapter 12. The efficiency curves (Figure 15.8) demonstrate this effect. The more threads there are, the more chunks there are (with a guided schedule), the more `run()` method calls there are, the sooner the JVM can detect the hot spot and compile it, the sooner the program can start running fast machine code, and the greater the speedup and efficiency. The efficiency curves increase until about $K = 4$ or 5, after which there is no further improvement.

While the curves for the parallel program without reduction (Figure 15.7) do show a slight superlinear speedup until about $K = 3$, as more processors are added the speedup becomes sublinear again. This is because the thread synchronization overhead, from more and more threads contending to access the one shared histogram variable, is wiping out the performance gain from the JIT compiler effect.

Although the reduction pattern improved the program's performance, it did increase the program's memory usage. Instead of one copy of the histogram array, the program now has $K$+1 copies: the shared reduction variable, and $K$ per-thread variables in the $K$ threads. The original program uses 4,004 bytes of storage for the histogram array (1,001 array elements times 4 bytes per `int` array element). On 8 processors, the program with reduction uses 36,036 bytes of storage for all the copies of the histogram array (not counting the extra padding to avert cache interference). In a different program with more data, $K$+1 copies of the data structure might require a substantial amount of memory. Increased storage is the price we pay for the parallel reduction pattern's benefit of reduced running time.

## 15.7 Critical Sections

While many programs that need parallel reduction can use reduction operators, sometimes the reduction computation is more complicated than combining the reduction variable and the per-thread variables with a binary operation. In that case, the program must synchronize the threads when executing an arbitrary block of code to perform the reduction. Parallel Java provides the **critical section** for this purpose.

To execute a critical section inside a parallel region, call the `critical()` method, passing in a ParallelSection whose `run()` method contains the code for the critical section.

```
new ParallelTeam().execute (new ParallelRegion()
   {
   public void run()
      {
      . . .
      critical (new ParallelSection()
         {
         public void run()
            {
            // Code for the critical section
            }
         });
      . . .
      }
   });
```

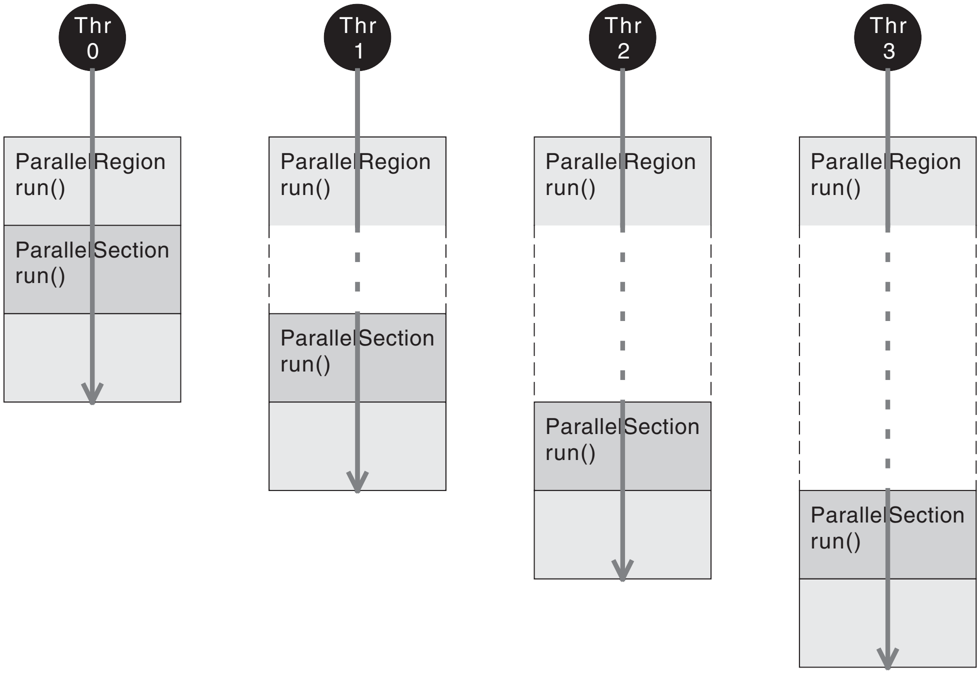


**Figure 15.9** Critical section

When each parallel team thread calls the `critical()` method, the threads automatically coordinate with each other so that only one thread at a time executes the critical section (Figure 15.9). If one thread is in the middle of a `critical()` method call, and another thread calls the `critical()` method, the second thread blocks. Any further threads calling the `critical()` method also block. When the first thread returns from the `critical()` method, one of the waiting threads unblocks and proceeds to execute the code in the critical section. We say the threads are executing the critical section in a **mutually exclusive** fashion. When a particular thread's turn to execute the critical section arrives, the thread calls the ParallelSection's `run()` method. Thus, only one thread at a time calls the `run()` method. When the `run()` method returns, the thread continues executing whatever comes after the critical section in the parallel region.

## 15.8 Parallel Version with Critical Section

The third version of the parallel Mandelbrot Set histogram program uses the reduction pattern, with the reduction computation in a critical section. As in the second version, each thread computes the histogram for a portion of the pixels and stores the histogram data in a per-thread variable. Code in the critical section combines the per-thread histograms to form the complete histogram stored in a shared variable. Here is the source code for class MSHistogramSmp3.

```
package edu.rit.smp.fractal;
import edu.rit.pj.IntegerForLoop;
import edu.rit.pj.ParallelRegion;
import edu.rit.pj.ParallelSection;
import edu.rit.pj.ParallelTeam;
import java.io.BufferedWriter;
import java.io.File;
import java.io.FileWriter;
import java.io.PrintWriter;
public class MSHistogramSmp3
   {
   // Command line arguments.
   static int width;
   static int height;
   static double xcenter;
   static double ycenter;
   static double resolution;
   static int maxiter;
   static File outfile;

   // Initial pixel offsets from center.
   static int xoffset;
   static int yoffset;
```

Because the critical section synchronizes the parallel team threads, the shared reduction variable does not need to be multiple-thread safe, and type `int[]` suffices.

```
   // Histogram (array of counters indexed by pixel value).
   static int[] histogram;

   /**
    * Mandelbrot Set main program.
    */
   public static void main
      (String[] args)
      throws Exception
      {
      // Start timing.
      long t1 = System.currentTimeMillis();

      // Validate command line arguments.
      if (args.length != 7) usage();
      width = Integer.parseInt (args[0]);
      height = Integer.parseInt (args[1]);
      xcenter = Double.parseDouble (args[2]);
      ycenter = Double.parseDouble (args[3]);
      resolution = Double.parseDouble (args[4]);
      maxiter = Integer.parseInt (args[5]);
      outfile = new File (args[6]);

      // Initial pixel offsets from center.
      xoffset = -(width - 1) / 2;
      yoffset = (height - 1) / 2;

      // Create histogram.
      histogram = new int [maxiter + 1];

      long t2 = System.currentTimeMillis();

      // Parallel computation region.
      new ParallelTeam().execute (new ParallelRegion()
         {
         public void run() throws Exception
            {
            execute (0, height-1, new IntegerForLoop()
               {
```

The per-thread variables, with the extra padding to avert cache interference, are the same as in the second version.

```
               // Per-thread histogram, plus extra padding.
               int[] thr_histogram = new int [maxiter + 1 + 32];
               long p0, p1, p2, p3, p4, p5, p6, p7;
               long p8, p9, pa, pb, pc, pd, pe, pf;

               public void run (int first, int last)
                  {
                  // Compute all rows and columns.
                  for (int r = first; r <= last; ++ r)
                     {
                     double y =
                        ycenter + (yoffset - r) / resolution;

                     for (int c = 0; c < width; ++ c)
                        {
                        double x =
                           xcenter + (xoffset + c) / resolution;

                        // Iterate until convergence.
                        int i = 0;
```

```
         double aold = 0.0;
         double bold = 0.0;

         double a = 0.0;
         double b = 0.0;
         double zmagsqr = 0.0;
         while (i < maxiter && zmagsqr <= 4.0)
            {
            ++ i;
            a = aold*aold - bold*bold + x;
            b = 2.0*aold*bold + y;
            zmagsqr = a*a + b*b;
            aold = a;
            bold = b;
            }
```

Inside the parallel for loop, each thread increments its own per-thread histogram counter, the same as in the second version.

```
         // Increment histogram counter.
         ++ thr_histogram[i];
         }
      }
   }
```

This time, instead of a reduction operator, code in a critical section adds the per-thread histogram into the shared reduction variable. The `critical()` method must be invoked on the ParallelRegion object, but this code is inside the IntegerForLoop object. The `region()` method returns a reference to the ParallelRegion within which this IntegerForLoop is executing.

```
// Reduce per-thread histogram into global
// histogram.
public void finish() throws Exception
   {
   region().critical (new ParallelSection()
      {
      public void run()
         {
         for (int i = 0; i <= maxiter; ++ i)
            {
            histogram[i] += thr_histogram[i];
            }
         }
      });
```

```
                  }
               });
            }
         });

      long t3 = System.currentTimeMillis();

      // Print histogram.
      PrintWriter out =
         new PrintWriter
            (new BufferedWriter
               (new FileWriter (outfile)));
      for (int i = 0; i <= maxiter; ++ i)
         {
         out.print (i);
         out.print ('\t');
         out.print (histogram[i]);
         out.println();
         }
      out.close();

      // Stop timing.
      long t4 = System.currentTimeMillis();
      System.out.println ((t2-t1) + " msec pre");
      System.out.println ((t3-t2) + " msec calc");
      System.out.println ((t4-t3) + " msec post");
      System.out.println ((t4-t1) + " msec total");
      }
   }
```

What about performance? For the Mandelbrot Set histogram problem, the parallel version with a reduction operator (MSHistogramSmp2) and the parallel version with a critical section (MSHistogramSmp3) have running times that differ by only a few milliseconds, well within the expected measurement error. Both parallel reduction techniques yield essentially the same performance for this problem.

## 15.9 Summary: Combining Partial Results

To sum up, we've now seen four ways to design an SMP parallel in which multiple threads each compute part of the answer, and the partial results are combined to yield the complete answer stored in a shared variable. For different programs one technique might be more applicable than another; it's useful to know them all.

1. **No reduction pattern**. Each thread updates the shared variable directly using a multiple-thread-safe operation. This approach might reduce the program's performance if the updates are frequent.

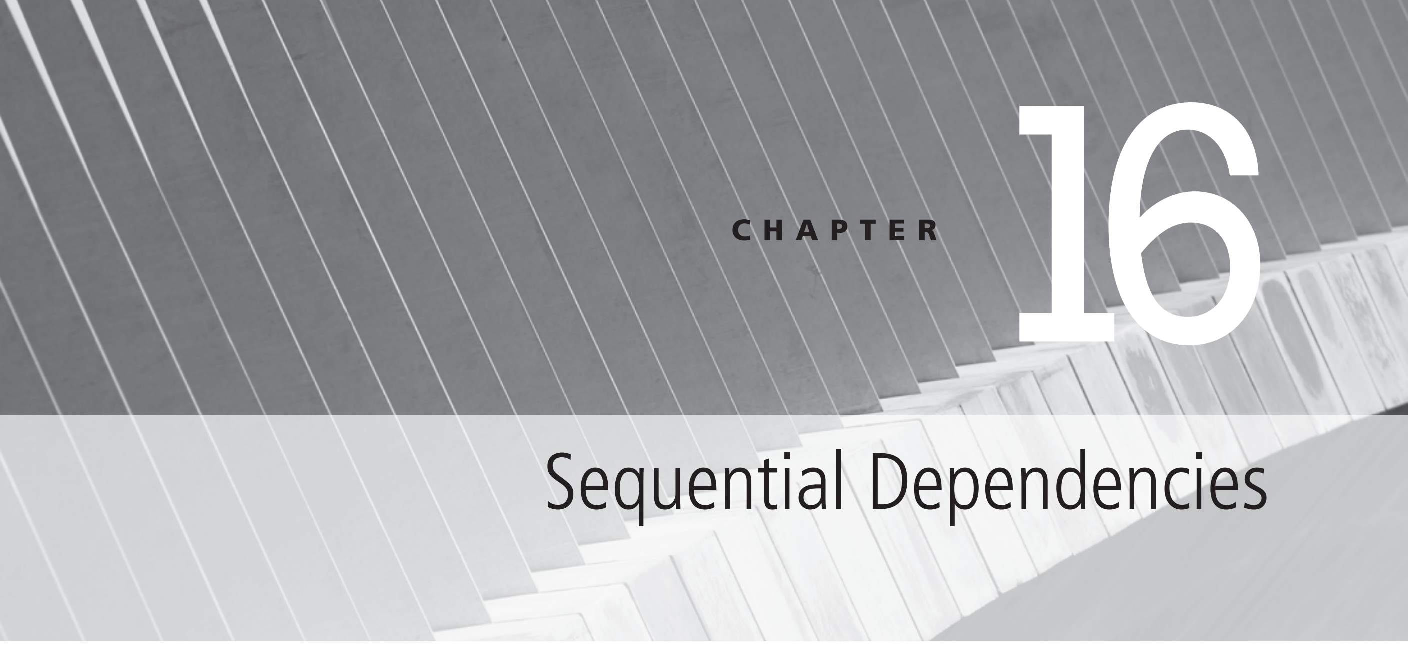
2. **Reduction pattern**. Each thread accumulates its partial result in its own per-thread variable (which does not have to be multiple-thread safe). As its last act, each thread updates the shared variable using a multiple-thread-safe operation. This approach might improve the program's performance if updates to the threads' partial results are frequent.
3. **Reduction pattern with reduction operator**. Each thread accumulates its partial result in its own per-thread variable. As its last act, each thread updates the shared variable using a reduction operator.
4. **Reduction pattern with critical section**. Each thread accumulates its partial result in its own per-thread variable. As its last act, each thread updates the shared variable in a critical section. In this case the shared variable does not have to be multiple-thread safe. This approach is appropriate when combining the partial results requires executing a whole section of code, not just a reduction operator.

**Table 15.1** MSHistogramSeq/MSHistogramSmp running-time metrics

| *N* | *K* | *T* | *Spdup* | *Eff* | *EDSF* | *N* | *K* | *T* | *Spdup* | *Eff* | *EDSF* |
|---:|---:|---:|---:|---:|---:|---:|---:|---:|---:|---:|---:|
| 10M | seq | 45148 | | | | 80M | seq | 353413 | | | |
| 10M | 1 | 45590 | 0.990 | 0.990 | | 80M | 1 | 356381 | 0.992 | 0.992 | |
| 10M | 2 | 22375 | 2.018 | 1.009 | -0.018 | 80M | 2 | 174571 | 2.024 | 1.012 | -0.020 |
| 10M | 3 | 14685 | 3.074 | 1.025 | -0.017 | 80M | 3 | 114366 | 3.090 | 1.030 | -0.019 |
| 10M | 4 | 11205 | 4.029 | 1.007 | -0.006 | 80M | 4 | 86936 | 4.065 | 1.016 | -0.008 |
| 10M | 5 | 9120 | 4.950 | 0.990 | 0.000 | 80M | 5 | 70189 | 5.035 | 1.007 | -0.004 |
| 10M | 6 | 7863 | 5.742 | 0.957 | 0.007 | 80M | 6 | 61051 | 5.789 | 0.965 | 0.006 |
| 10M | 7 | 7007 | 6.443 | 0.920 | 0.013 | 80M | 7 | 53115 | 6.654 | 0.951 | 0.007 |
| 10M | 8 | 6345 | 7.116 | 0.889 | 0.016 | 80M | 8 | 47922 | 7.375 | 0.922 | 0.011 |
| 20M | seq | 88399 | | | | 160M | seq | 721166 | | | |
| 20M | 1 | 89205 | 0.991 | 0.991 | | 160M | 1 | 727179 | 0.992 | 0.992 | |
| 20M | 2 | 43730 | 2.021 | 1.011 | -0.020 | 160M | 2 | 356261 | 2.024 | 1.012 | -0.020 |
| 20M | 3 | 28657 | 3.085 | 1.028 | -0.018 | 160M | 3 | 231504 | 3.115 | 1.038 | -0.022 |
| 20M | 4 | 21847 | 4.046 | 1.012 | -0.007 | 160M | 4 | 176971 | 4.075 | 1.019 | -0.009 |
| 20M | 5 | 17735 | 4.984 | 0.997 | -0.001 | 160M | 5 | 144561 | 4.989 | 0.998 | -0.002 |
| 20M | 6 | 15270 | 5.789 | 0.965 | 0.005 | 160M | 6 | 124176 | 5.808 | 0.968 | 0.005 |
| 20M | 7 | 13595 | 6.502 | 0.929 | 0.011 | 160M | 7 | 107261 | 6.723 | 0.960 | 0.005 |
| 20M | 8 | 12141 | 7.281 | 0.910 | 0.013 | 160M | 8 | 99002 | 7.284 | 0.911 | 0.013 |
| 40M | seq | 180337 | | | | 320M | seq | 1413394 | | | |
| 40M | 1 | 181903 | 0.991 | 0.991 | | 320M | 1 | 1424946 | 0.992 | 0.992 | |
| 40M | 2 | 89114 | 2.024 | 1.012 | -0.020 | 320M | 2 | 697856 | 2.025 | 1.013 | -0.021 |
| 40M | 3 | 58349 | 3.091 | 1.030 | -0.019 | 320M | 3 | 458947 | 3.080 | 1.027 | -0.017 |
| 40M | 4 | 44457 | 4.056 | 1.014 | -0.007 | 320M | 4 | 346455 | 4.080 | 1.020 | -0.009 |
| 40M | 5 | 36008 | 5.008 | 1.002 | -0.003 | 320M | 5 | 283425 | 4.987 | 0.997 | -0.001 |
| 40M | 6 | 30879 | 5.840 | 0.973 | 0.004 | 320M | 6 | 243228 | 5.811 | 0.968 | 0.005 |
| 40M | 7 | 27784 | 6.491 | 0.927 | 0.012 | 320M | 7 | 214852 | 6.578 | 0.940 | 0.009 |
| 40M | 8 | 24697 | 7.302 | 0.913 | 0.012 | 320M | 8 | 195240 | 7.239 | 0.905 | 0.014 |

**Table 15.2** MSHistogramSeq/MSHistogramSmp2 running-time metrics

| *N* | *K* | *T* | *Spdup* | *Eff* | *EDSF* | *N* | *K* | *T* | *Spdup* | *Eff* | *EDSF* |
|---:|---:|---:|---:|---:|---:|---:|---:|---:|---:|---:|---:|
| 10M | seq | 45148 | | | | 80M | seq | 353413 | | | |
| 10M | 1 | 45118 | 1.001 | 1.001 | | 80M | 1 | 352895 | 1.001 | 1.001 | |
| 10M | 2 | 21190 | 2.131 | 1.065 | -0.061 | 80M | 2 | 165245 | 2.139 | 1.069 | -0.063 |
| 10M | 3 | 13595 | 3.321 | 1.107 | -0.048 | 80M | 3 | 105732 | 3.343 | 1.114 | -0.051 |
| 10M | 4 | 10409 | 4.337 | 1.084 | -0.026 | 80M | 4 | 76391 | 4.626 | 1.157 | -0.045 |
| 10M | 5 | 7957 | 5.674 | 1.135 | -0.030 | 80M | 5 | 61650 | 5.733 | 1.147 | -0.032 |
| 10M | 6 | 6636 | 6.803 | 1.134 | -0.024 | 80M | 6 | 51442 | 6.870 | 1.145 | -0.025 |
| 10M | 7 | 5950 | 7.588 | 1.084 | -0.013 | 80M | 7 | 43196 | 8.182 | 1.169 | -0.024 |
| 10M | 8 | 5107 | 8.840 | 1.105 | -0.013 | 80M | 8 | 39449 | 8.959 | 1.120 | -0.015 |
| 20M | seq | 88399 | | | | 160M | seq | 721166 | | | |
| 20M | 1 | 88305 | 1.001 | 1.001 | | 160M | 1 | 720058 | 1.002 | 1.002 | |
| 20M | 2 | 41404 | 2.135 | 1.068 | -0.062 | 160M | 2 | 337074 | 2.139 | 1.070 | -0.064 |
| 20M | 3 | 26524 | 3.333 | 1.111 | -0.049 | 160M | 3 | 212614 | 3.392 | 1.131 | -0.057 |
| 20M | 4 | 20295 | 4.356 | 1.089 | -0.027 | 160M | 4 | 155842 | 4.628 | 1.157 | -0.045 |
| 20M | 5 | 15503 | 5.702 | 1.140 | -0.031 | 160M | 5 | 129859 | 5.553 | 1.111 | -0.025 |
| 20M | 6 | 12937 | 6.833 | 1.139 | -0.024 | 160M | 6 | 107954 | 6.680 | 1.113 | -0.020 |
| 20M | 7 | 10880 | 8.125 | 1.161 | -0.023 | 160M | 7 | 90609 | 7.959 | 1.137 | -0.020 |
| 20M | 8 | 9918 | 8.913 | 1.114 | -0.014 | 160M | 8 | 82564 | 8.735 | 1.092 | -0.012 |
| 40M | seq | 180337 | | | | 320M | seq | 1413394 | | | |
| 40M | 1 | 180094 | 1.001 | 1.001 | | 320M | 1 | 1410755 | 1.002 | 1.002 | |
| 40M | 2 | 84373 | 2.137 | 1.069 | -0.063 | 320M | 2 | 660569 | 2.140 | 1.070 | -0.064 |
| 40M | 3 | 54006 | 3.339 | 1.113 | -0.050 | 320M | 3 | 422585 | 3.345 | 1.115 | -0.051 |
| 40M | 4 | 41312 | 4.365 | 1.091 | -0.027 | 320M | 4 | 312725 | 4.520 | 1.130 | -0.038 |
| 40M | 5 | 31495 | 5.726 | 1.145 | -0.031 | 320M | 5 | 254333 | 5.557 | 1.111 | -0.025 |
| 40M | 6 | 26266 | 6.866 | 1.144 | -0.025 | 320M | 6 | 211378 | 6.687 | 1.114 | -0.020 |
| 40M | 7 | 22079 | 8.168 | 1.167 | -0.024 | 320M | 7 | 177334 | 7.970 | 1.139 | -0.020 |
| 40M | 8 | 20172 | 8.940 | 1.117 | -0.015 | 320M | 8 | 155797 | 9.072 | 1.134 | -0.017 |



# CHAPTER 16

# Sequential Dependencies

in which we learn how to analyze an algorithm for sequential dependencies; we learn how to parallelize the algorithm while enforcing the sequential dependencies; and we discover how the CPU's cache memory affects a parallel program's performance

# CHAPTER 16 Sequential Dependencies

## 16.1 Floyd's Algorithm

Suppose you are making a map. You have marked several cities on the map. Some of the cities are connected by roads, and you have marked these on the map as well. Some roads cross over each other, but the roads connect only to the cities. You know each road's distance in kilometers. Your task is to prepare a table of the distances between each pair of cities. The table lists starting cities along the left and destination cities along the top. The table entry for starting city *X*, destination city *Y* gives the shortest total distance one has to travel to get from *X* to *Y* following the roads. This is the **all-pairs shortest-paths problem**.

A **graph**, such as the one in Figure 16.1, consists of a set of **vertices**, or nodes (the circles), and a set of **edges** connecting pairs of vertices (the lines). Two vertices connected by an edge are said to be **adjacent**. A **path** from some vertex *X* to some vertex *Y* consists of a sequence of vertices where each vertex is adjacent to its predecessor. For example, in Figure 16.1, *AHGCB* is one path from vertex *A* to vertex *B*. (This is not the only path from *A* to *B*.) A **weight** (a number) is associated with each edge. The **length** of a path is the sum of the weights of the edges in the path. The all-pairs shortest-paths problem is to find, for each starting vertex *X* and each ending vertex *Y*, the length of the shortest path from *X* to *Y*. Identifying the map as a graph, the cities as vertices, the roads as edges, and the road distances as edge weights transforms the problem of preparing the map distance table into the graph all-pairs shortest-paths problem.

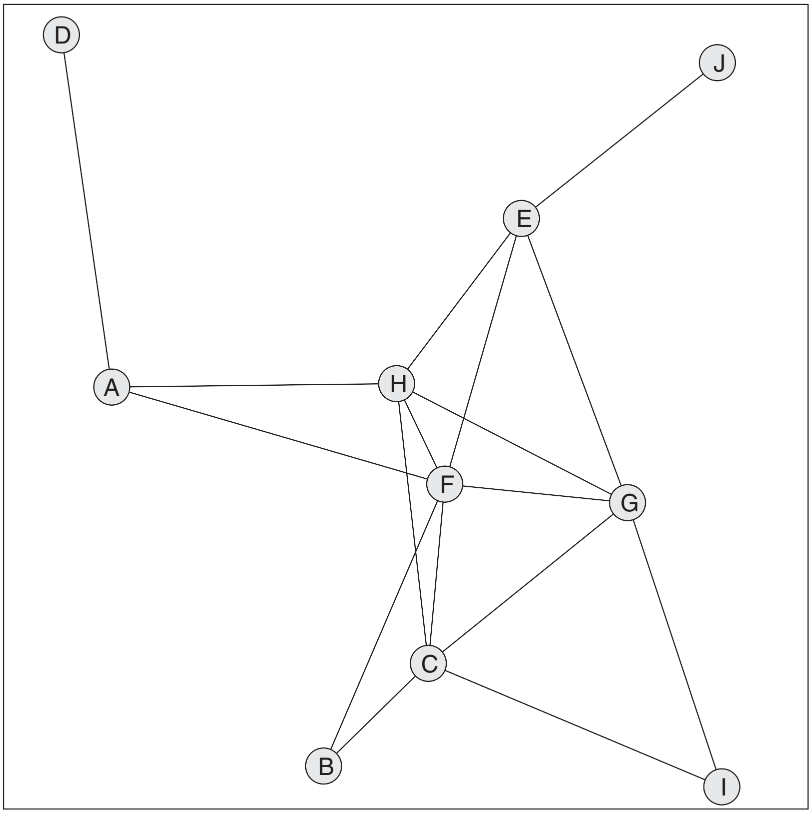


**Figure 16.1** A graph

| | *A* | *B* | *C* | *D* | *E* | *F* | *G* | *H* | *I* | *J* |
|---|---|---|---|---|---|---|---|---|---|---|
| ***A*** | 0 | ∞ | ∞ | 462 | ∞ | 451 | ∞ | 370 | ∞ | ∞ |
| ***B*** | ∞ | 0 | 190 | ∞ | ∞ | 399 | ∞ | ∞ | ∞ | ∞ |
| ***C*** | ∞ | 190 | 0 | ∞ | ∞ | 234 | 333 | 366 | 414 | ∞ |
| ***D*** | 462 | ∞ | ∞ | 0 | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ |
| ***E*** | ∞ | ∞ | ∞ | ∞ | 0 | 359 | 394 | 269 | ∞ | 325 |
| ***F*** | 451 | 399 | 234 | ∞ | 359 | 0 | 239 | 144 | ∞ | ∞ |
| ***G*** | ∞ | ∞ | 333 | ∞ | 394 | 239 | 0 | 337 | 389 | ∞ |
| ***H*** | 370 | ∞ | 366 | ∞ | 269 | 144 | 337 | 0 | ∞ | ∞ |
| ***I*** | ∞ | ∞ | 414 | ∞ | ∞ | ∞ | 389 | ∞ | 0 | ∞ |
| ***J*** | ∞ | ∞ | ∞ | ∞ | 325 | ∞ | ∞ | ∞ | ∞ | 0 |

**Figure 16.2** Distance matrix

One way to represent a graph is a **weight matrix**, or **distance matrix**. The distance matrix element at row $r$ and column $c$, $d_{rc}$, is the weight of the edge between vertex $r$ and vertex $c$. If vertices $r$ and $c$ are not adjacent, then $d_{rc}$ is infinity ($\infty$). Figure 16.2 gives the distance matrix for the graph in Figure 16.1. These distances were derived by placing the vertices at random locations in a square of side 1,000 units and measuring the Euclidean distances between adjacent vertices. However, in general, the edge weights could be any numbers, not necessarily Euclidean distances. This particular distance matrix is *symmetric;* the distance from vertex $r$ to vertex $c$ is the same as from $c$ to $r$. Again, in general, the distance matrix need not be symmetric.

| | ***A*** | ***B*** | ***C*** | ***D*** | ***E*** | ***F*** | ***G*** | ***H*** | ***I*** | ***J*** |
|---|---|---|---|---|---|---|---|---|---|---|
| ***A*** | 0 | 850 | 685 | 462 | 639 | 451 | 690 | 370 | 1079 | 964 |
| ***B*** | 850 | 0 | 190 | 1312 | 758 | 399 | 523 | 543 | 604 | 1083 |
| ***C*** | 685 | 190 | 0 | 1147 | 593 | 234 | 333 | 366 | 414 | 918 |
| ***D*** | 462 | 1312 | 1147 | 0 | 1101 | 913 | 1152 | 832 | 1541 | 1426 |
| ***E*** | 639 | 758 | 593 | 1101 | 0 | 359 | 394 | 269 | 783 | 325 |
| ***F*** | 451 | 399 | 234 | 913 | 359 | 0 | 239 | 144 | 628 | 684 |
| ***G*** | 690 | 523 | 333 | 1152 | 394 | 239 | 0 | 337 | 389 | 719 |
| ***H*** | 370 | 543 | 366 | 832 | 269 | 144 | 337 | 0 | 726 | 594 |
| ***I*** | 1079 | 604 | 414 | 1541 | 783 | 628 | 389 | 726 | 0 | 1108 |
| ***J*** | 964 | 1083 | 918 | 1426 | 325 | 684 | 719 | 594 | 1108 | 0 |

**Figure 16.3** Distance matrix after running Floyd's Algorithm

In 1962, Robert Floyd published an algorithm for the all-pairs shortest-paths problem. The input to Floyd's Algorithm is the $n \times n$ distance matrix $d$ for a graph of $n$ vertices. On output, the contents of $d$ have been replaced such that $d_{rc}$ is the length of the shortest path from vertex $r$ to vertex $c$. If there is no path from $r$ to $c$, $d_{rc}$ is $\infty$. Floyd's Algorithm is quite simple:

```
for $i = 0$ to $n-1$
    for $r = 0$ to $n-1$
        for $c = 0$ to $n-1$
            $d_{rc} \leftarrow \min(d_{rc}, d_{ri} + d_{ic})$
```

Because Floyd's Algorithm has three nested loops each with $n$ iterations, the algorithm's running time $T$ is $O(n^3)$, and the problem size $N$ is $n^3$.

Figure 16.3 shows the result when Floyd's Algorithm is run on the distance matrix in Figure 16.2. The shortest path from vertex $A$ to vertex $B$ has a length of 850. In its present form, Floyd's Algorithm only gives the *length* of the shortest path, not the shortest path itself. Examining the original distance matrix shows that the path from $A$ to $B$ of length 850 is path $AFB$ with edge weights of 451 and 399.

## 16.2 Input and Output Files

Before we can write a program for Floyd's Algorithm, we must define how the program will obtain the input distance matrix. The program will read the input distance matrix from a file and write the output distance matrix to another file, using class edu.rit.io.DoubleMatrixFile from the Parallel Java Library to do the actual I/O.

To store a matrix of type double in a file, construct an instance of class DoubleMatrixFile, specifying the number of rows, the number of columns, and the matrix to be written.

```
double[][] matrix = new double [R] [C];
DoubleMatrixFile dmf = new DoubleMatrixFile (R, C, matrix);
```

Then call the double matrix file's `prepareToWrite()` method, specifying the output stream on which to write the matrix.

```
DoubleMatrixFile.Writer writer = dmf.prepareToWrite (outstream);
```

The `prepareToWrite()` method returns a writer object, an instance of class DoubleMatrixFile.Writer. Call the writer's `write()` method to write the matrix, and then close the writer.

```
writer.write();
writer.close();
```

Look familiar? This is the same pattern that class edu.rit.image.PJGImage uses to write a PJG image file. The double matrix file writer stores the matrix in a binary format, where each element of type `double` occupies 8 bytes. For further information about the double matrix file format, refer to the documentation for class DoubleMatrixFile.

To read the matrix back in from the file, do the following:

```
DoubleMatrixFile dmf = new DoubleMatrixFile();
DoubleMatrixFile.Reader reader = dmf.prepareToRead (instream);
reader.read();
reader.close();
int R = dmf.getRowCount();
int C = dmf.getColCount();
double[][] matrix = dmf.getMatrix();
```

After executing the preceding code, `R` is the number of rows in the matrix that was read in, `C` is the number of columns, and `matrix` is a reference to the matrix itself.

Here is a short program to generate a random distance matrix file to feed into Floyd's Algorithm. The program's arguments are a random seed, an adjacency radius, the number of vertices *n*, and the distance matrix file name. The program generates *n* vertices at random locations in the unit square. Two vertices are adjacent if the Euclidean distance between them is less than or equal to the adjacency radius, in which case the edge weight is the Euclidean distance.

```
package edu.rit.smp.network;
import edu.rit.io.DoubleMatrixFile;
import edu.rit.util.Random;
import java.io.BufferedOutputStream;
import java.io.FileOutputStream;
public class FloydRandom
   {
   public static void main
      (String[] args)
      throws Exception
      {
      // Parse command line arguments.
      if (args.length != 4) usage();
      long seed = Long.parseLong (args[0]);
      double radius = Double.parseDouble (args[1]);
      int n = Integer.parseInt (args[2]);
      String matrixfile = args[3];

      // Set up pseudorandom number generator.
      Random prng = Random.getInstance (seed);

      // Generate random node locations in the unit square.
      double[] x = new double [n];
      double[] y = new double [n];
      for (int i = 0; i < n; ++ i)
         {
         x[i] = prng.nextDouble();
         y[i] = prng.nextDouble();
         }

      // Compute distance matrix elements.
      double[][] d = new double [n] [n];
      for (int r = 0; r < n; ++ r)
         {
         double[] d_r = d[r];
         for (int c = 0; c < n; ++ c)
            {
            double dx = x[r] - x[c];
            double dy = y[r] - y[c];
            double distance = Math.sqrt (dx*dx + dy*dy);
            d_r[c] =
               (distance <= radius ?
                  distance :
                  Double.POSITIVE_INFINITY);
            }
         }
      // Write distance matrix to output file.
      DoubleMatrixFile dmf = new DoubleMatrixFile (n, n, d);
      DoubleMatrixFile.Writer writer =
         dmf.prepareToWrite
            (new BufferedOutputStream
               (new FileOutputStream (matrixfile)));
      writer.write();
      writer.close();
      }
   }
```

## 16.3 Sequential Program

Here is the source code for class FloydSeq, a sequential version of Floyd's Algorithm. The program's command-line arguments are the name of the input distance matrix file and the name of the output distance matrix file.

```
package edu.rit.smp.network;
import edu.rit.io.DoubleMatrixFile;
import java.io.BufferedInputStream;
import java.io.BufferedOutputStream;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
public class FloydSeq
   {
   // Number of nodes.
   static int n;

   // Distance matrix.
   static double[][] d;

   /**
    * Main program.
    */
   public static void main
      (String[] args)
      throws Throwable
      {
      // Start timing.
      long t1 = System.currentTimeMillis();
      // Parse command line arguments.
      if (args.length != 2) usage();
      File infile = new File (args[0]);
      File outfile = new File (args[1]);

      // Read distance matrix from input file.
      DoubleMatrixFile dmf = new DoubleMatrixFile();
      DoubleMatrixFile.Reader reader =
         dmf.prepareToRead
            (new BufferedInputStream
               (new FileInputStream (infile)));
      reader.read();
      reader.close();
      n = dmf.getRowCount();
      d = dmf.getMatrix();

      // Run Floyd's Algorithm.
      //     for i = 0 to N-1
      //         for r = 0 to N-1
      //             for c = 0 to N-1
      //                 D[r,c] = min (D[r,c], D[r,i] + D[i,c])
      long t2 = System.currentTimeMillis();
      for (int i = 0; i < n; ++ i)
         {
         double[] d_i = d[i];
         for (int r = 0; r < n; ++ r)
            {
            double[] d_r = d[r];
            for (int c = 0; c < n; ++ c)
               {
               d_r[c] = Math.min (d_r[c], d_r[i] + d_i[c]);
               }
            }
         }
      long t3 = System.currentTimeMillis();

      // Write distance matrix to output file.
      DoubleMatrixFile.Writer writer =
         dmf.prepareToWrite
            (new BufferedOutputStream
               (new FileOutputStream (outfile)));
      writer.write();
      writer.close();
      // Stop timing.
      long t4 = System.currentTimeMillis();
      System.out.println ((t2-t1) + " msec pre");
      System.out.println ((t3-t2) + " msec calc");
      System.out.println ((t4-t3) + " msec post");
      System.out.println ((t4-t1) + " msec total");
      }
   }
```

## 16.4 Parallelizing Floyd's Algorithm

Until now, all the programs we've studied had an outermost loop with no sequential dependencies between the loop iterations; the iterations could be done in any order. Thus, to get each program's parallel version, we simply changed the outermost loop to a parallel loop, and let the parallel loop divide the loop iterations among the parallel team threads however it pleased. However, to parallelize Floyd's Algorithm we must be more careful.

Let's analyze Floyd's Algorithm's three nested loops one at a time. We'll assume that the distance from a vertex to itself is always 0. That is, the distance matrix's diagonal elements are all 0: $d_{ii} = 0$ for all $i$.

Can we do the iterations of the outer loop, the loop over $i$, in parallel? During iteration $i$, we will store a value into every distance matrix element $d_{rc}$, and the value that is stored depends on the current values of $d_{rc}$, $d_{ri}$, and $d_{ic}$. However, on the *previous* iteration the values of $d_{rc}$, $d_{ri}$, and $d_{ic}$ could have changed, because the same thing happened during the previous iteration—a value was stored into each element. Therefore, we have a sequential dependency from each iteration $i$ to the next iteration $i$+1. We cannot do iterations for different values of $i$ in parallel; we must do them in sequence. This means we cannot make the outer loop a parallel loop; it must remain a plain sequential loop.

If not the outer loop, can we do the iterations of the middle loop over $r$ in parallel? For a given outer loop iteration $i$, the middle loop iterates over all the distance matrix rows. At first glance, it looks like we must do the middle loop iterations in sequence: During the middle loop iteration where $r = i$, values are stored into the elements in row $i$, and these values affect the calculation of elements in higher rows, which depend on elements in row $i$ ($d_{ic}$). However, notice that during the middle loop iteration where $r = i$, the last line of the algorithm sets the elements in row $i$ as follows (substituting $i$ for $r$ in the subscripts):

$$d_{ic} \leftarrow \min\left(d_{ic},\ d_{ii} + d_{ic}\right)$$

Now, because $d_{ii}$ is 0 (we assume), $d_{ii} + d_{ic}$ equals $d_{ic}$, so the right side of the preceding assignment is $d_{ic}$, and $d_{ic}$ is set to itself. Thus, during outer loop iteration $i$, the elements in row $i$ do not change their values. By similar reasoning, during outer loop iteration $i$ the elements in column $i$ do not change their values. In general, then, during outer loop iteration $i$ the new value of each element $d_{rc}$ depends on its old value, an element in column $i$ which is constant ($d_{ri}$), and an element in row $i$ which is constant ($d_{ic}$), but $d_{rc}$ does not depend on any other element. Therefore, during outer loop iteration $i$, the order in which rows are calculated does *not* in fact matter, and the middle loop *can* become a parallel loop.

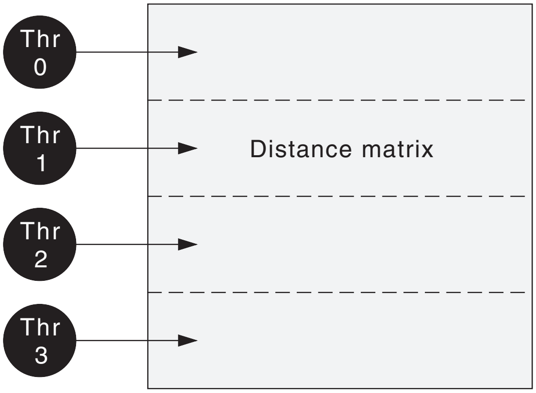


**Figure 16.4** Distance matrix sliced by rows

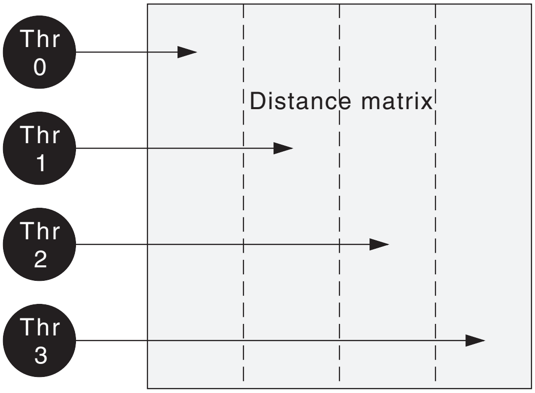


**Figure 16.5** Distance matrix sliced by columns

The parallel version of Floyd's Algorithm is the following:

```
for $i = 0$ to $n–1$
    parallel for $r = 0$ to $n–1$
        for $c = 0$ to $n–1$
            $d_{rc} \leftarrow \min(d_{rc}, d_{ri} + d_{ic})$
```

Because every middle loop iteration does the same amount of computation, dividing the middle loop iterations equally among the parallel team threads using a fixed schedule yields a balanced load. The preceding algorithm slices the distance matrix by rows (Figure 16.4). Each thread does every outer loop iteration; within each outer loop iteration, each thread computes a subset of the rows and all the columns.

It would also be possible to make the inner loop, the loop over *c*, a parallel loop:

```
for $i = 0$ to $n–1$
    for $r = 0$ to $n–1$
        parallel for $c = 0$ to $n–1$
            $d_{rc} \leftarrow \min(d_{rc}, d_{ri} + d_{ic})$
```

This version slices the distance matrix by columns (Figure 16.5). Each thread does every outer loop iteration; within each outer loop iteration, each thread computes all the rows and a subset of the columns.

## 16.5 Parallel Program with Row Slicing

Here is the source code for class FloydSmpRow, an SMP parallel version of Floyd's Algorithm in which each parallel team thread computes a subset of the rows and all the columns of the distance matrix.

```
package edu.rit.smp.network;
import edu.rit.io.DoubleMatrixFile;
import edu.rit.pj.IntegerForLoop;
import edu.rit.pj.ParallelRegion;
import edu.rit.pj.ParallelTeam;
import java.io.BufferedInputStream;
import java.io.BufferedOutputStream;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
public class FloydSmpRow
   {
```

The number of vertices n and the distance matrix d are now shared variables. Because it is a WORM variable, n needs no synchronization. Accessing d while reading the input file and writing the output file requires no synchronization, because at those points only the main program thread is active. During the execution of Floyd's Algorithm, all the parallel team threads are reading and updating d. Nonetheless, a little later, we will see that synchronization is not needed for d either.

```
   // Number of nodes.
   static int n;

   // Distance matrix.
   static double[][] d;

   /**
    * Main program.
    */
   public static void main
      (String[] args)
      throws Exception
      {
      // Start timing.
      long t1 = System.currentTimeMillis();

      // Parse command line arguments.
      if (args.length != 2) usage();
      File infile = new File (args[0]);
      File outfile = new File (args[1]);

      // Read distance matrix from input file.
      DoubleMatrixFile dmf = new DoubleMatrixFile();
      DoubleMatrixFile.Reader reader =
         dmf.prepareToRead
            (new BufferedInputStream
               (new FileInputStream (infile)));
      reader.read();
      reader.close();
      n = dmf.getRowCount();
      d = dmf.getMatrix();
      // Run Floyd’s Algorithm.
      //     for i = 0 to N-1
      //         parallel for r = 0 to N-1
      //             for c = 0 to N-1
      //                 D[r,c] = min (D[r,c], D[r,i] + D[i,c])
      long t2 = System.currentTimeMillis();
```

Having read the input distance matrix file, now comes the computational heart of the program, the part we parallelize. We put the *entire* computation of the distance matrix—*all three* loops of Floyd’s Algorithm—inside the parallel region. This way, the program creates only one parallel team of threads and creates only one parallel region of code for them to execute. If we put the parallel team and parallel region inside the outer loop, then each time around the outer loop the parallel team would incur the overhead of starting up and terminating execution of the parallel region, and we want to avoid that.

```
      new ParallelTeam().execute (new ParallelRegion()
         {
         public void run() throws Exception
            {
```

As stated earlier, the outer loop on `i` must remain a plain sequential loop. Each thread executes all the outer loop iterations.

```
            for (int ii = 0; ii < n; ++ ii)
               {
```

The local variables `i` and `d_i` are declared `final` so that code in the IntegerForLoop subclass (an anonymous inner class) can use these variables’ values. In Java, an inner class cannot refer to a local variable in the enclosing code unless the local variable is declared `final`.

```
               final int i = ii;
               final double[] d_i = d[i];
```

The middle loop on `r` becomes a parallel for loop. Each thread executes a subset of the middle loop iterations and computes a subset of the distance matrix rows. Because Floyd’s Algorithm is inherently load balanced, the parallel for loop’s default fixed schedule suffices.

```
               execute (0, n-1, new IntegerForLoop()
                  {
                  public void run (int first, int last)
                     {
                     for (int r = first; r <= last; ++ r)
                        {
                        double[] d_r = d[r];
```

The inner loop on `c` remains a sequential loop. Each thread computes all the distance matrix columns in the thread's subset of the rows.

```
                           for (int c = 0; c < n; ++ c)
                              {
                              d_r[c] =
                                 Math.min (d_r[c], d_r[i]+d_i[c]);
                              }
```

Earlier, we claimed that thread synchronization is not required when accessing the distance matrix `d`. Here's why. During a particular outer loop iteration *i*, each element of `d` (except those in row *i* and in column *i*) is read and written by only one thread. Thus, the threads do not conflict when accessing these elements, and no synchronization is required.

Although the elements in row *i* and in column *i* are read and written by multiple threads, earlier we proved that these elements' values do not change during outer loop iteration *i*. More precisely, the new value written back into the element is the same as the element's current value. Therefore, it doesn't matter if one thread writes one of these elements while another thread is reading one of these elements, and no synchronization is required.

```
                           }
                        }
                     });
```

This marks the end of the middle parallel loop over *r*. When a thread reaches this point, the thread is poised to go on to the next iteration of the outer loop over *i*. However, because of the sequential dependency from each outer loop iteration to the next, none of the threads can be allowed to proceed past this point until all the threads have finished the previous outer loop iteration and have reached this point. Thus, at this point all the threads must wait at a barrier. As explained in Chapter 6, a parallel for loop in Parallel Java includes an implicit barrier wait at the end of the loop. Although some parallel programs do not need this barrier wait, in the Floyd's Algorithm parallel program this barrier wait is crucial; the program will not compute the correct answer without it.

```
               }
            }
         });
      long t3 = System.currentTimeMillis();
      // Write distance matrix to output file.
      DoubleMatrixFile.Writer writer =
         dmf.prepareToWrite
            (new BufferedOutputStream
               (new FileOutputStream (outfile)));
      writer.write();
      writer.close();

      // Stop timing.
      long t4 = System.currentTimeMillis();
      System.out.println ((t2-t1) + " msec pre");
      System.out.println ((t3-t2) + " msec calc");
      System.out.println ((t4-t3) + " msec post");
      System.out.println ((t4-t1) + " msec total");
      }
   }
```

Table 16.1 (at the end of the chapter) lists, and Figure 16.6 plots, the FloydSmpRow program's performance on the "parasite" SMP parallel computer. The running times are for the calculation portion only, not including the distance matrix file I/O. The program was run on distance matrices with the numbers of vertices $n$ and problem sizes $N = n^3$ listed.

| $n$ | $N$ | |
|---:|---:|---:|
| 1,000 | 1,000,000,000 | (1G) |
| 1,260 | 2,000,376,000 | (2G) |
| 1,590 | 4,019,679,000 | (4G) |
| 2,000 | 8,000,000,000 | (8G) |
| 2,520 | 16,003,008,000 | (16G) |
| 3,180 | 32,157,432,000 | (32G) |

Each distance matrix input file was created with this command, where `$n` stands for one of the preceding $n$ values.

```
$ java edu.rit.smp.network.FloydRandom 142857 0.25 $n in$n.dat
```

These performance curves are like nothing we've seen so far, certainly nothing like what Amdahl's Law predicts. Consider the efficiency curves. For $n$ = 1,000 vertices ($N$ = 1G), the efficiencies are in the 1.0–1.2 range, with a small downward trend as $K$, the number of processors, increases. The efficiencies are greater than 1 because of the JIT compiler effect. The downward trend is due to the program's sequential fraction. For larger values of $n$, the efficiencies start out the same; but at a certain point as $K$ increases, the efficiencies abruptly jump up to the 2.0–2.2 range! Furthermore, the greater the number of vertices, the larger the number of processors before reaching the jump in efficiency. How can the parallel version possibly be twice as efficient as the sequential version? What is going on?

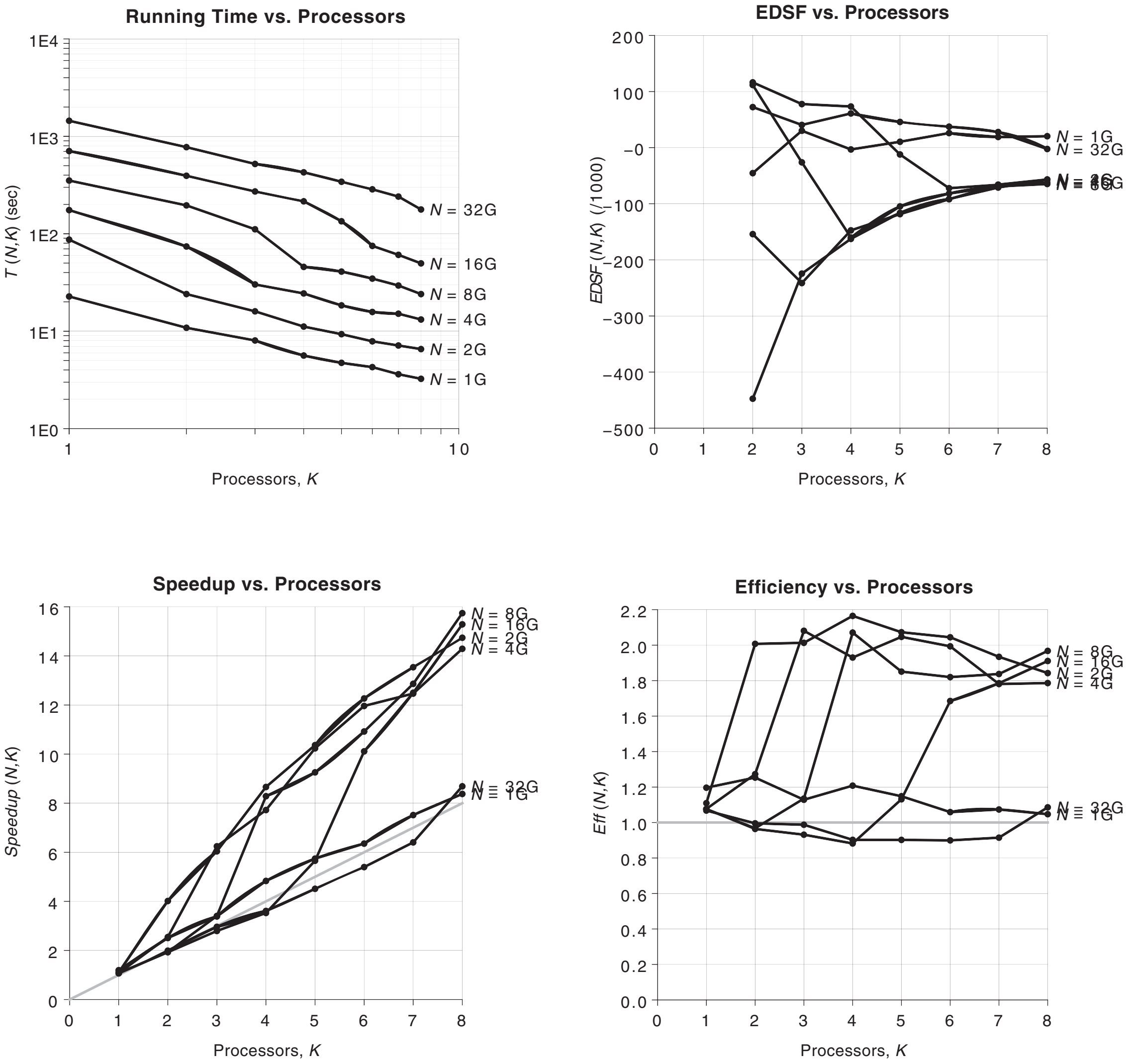


**Figure 16.6** FloydSeq/FloydSmpRow running-time metrics

The answer lies in this program's pattern of data accesses coupled with the behavior of the processors' cache memories.

First, let's calculate the amount of storage needed to hold each parallel team thread's portion of the distance matrix d, as a function of the number of vertices $n$ and the number of processors $K$. Because the distance matrix consists of $n \times n$ doubles and each double requires 8 bytes, d requires $8n^2$ bytes. But because the distance matrix is sliced equally among the threads, each processor accesses only $8n^2/K$ bytes. Here are the numbers of megabytes (1 megabyte = $2^{20}$ bytes) each processor accesses for the six $n$ values and for $K = 1$ to 8. The highlighted numbers show where the abrupt increase in efficiency occurs.

|  | *n* = | *n* = | *n* = | *n* = | *n* = | *n* = |
|---|---|---|---|---|---|---|
| *K* | *1,000* | *1,260* | *1,590* | *2,000* | *2,520* | *3,180* |
| 1 | 7.63 | 12.11 | 19.29 | 30.52 | 48.45 | 77.15 |
| 2 | 3.82 | 6.06 | 9.64 | 15.26 | 24.22 | 38.58 |
| 3 | 2.54 | 4.04 | 6.43 | 10.17 | 16.15 | 25.72 |
| 4 | 1.91 | 3.03 | 4.82 | 7.63 | 12.11 | 19.29 |
| 5 | 1.53 | 2.42 | 3.86 | 6.10 | 9.69 | 15.43 |
| 6 | 1.27 | 2.02 | 3.21 | 5.09 | 8.07 | 12.86 |
| 7 | 1.09 | 1.73 | 2.76 | 4.36 | 6.92 | 11.02 |
| 8 | 0.95 | 1.51 | 2.41 | 3.81 | 6.06 | 9.64 |

The “parasite” SMP parallel computer has four Sun Microsystems UltraSPARC-IV CPU chips. These are hyperthreaded CPUs each with two instruction units, letting each chip run two threads in parallel. The UltraSPARC-IV CPU also has a three-level cache. Each instruction unit has its own dedicated level-1 cache located on the CPU chip, with 64 kilobytes for instructions and 64 kilobytes for data. There is also a two-megabyte level-2 cache located on the CPU chip; it is not quite as fast as the level-1 cache. Finally, there is a 32-megabyte level-3 cache located outside the CPU chip; it is not as fast as the level-2 cache but faster than main memory. The level-2 and level-3 caches hold both instructions and data.

Consider what happens when the sequential Floyd’s Algorithm program (FloydSeq) runs with $n$ = 1,000 vertices. Once the program has started executing the three nested loops and the JIT compiler has had a chance to run, the machine instructions for the three nested loops have all been read into the cache from main memory, as have the machine instructions for the JVM itself. As the program executes the first outer loop iteration with $i = 0$, the program reads and writes every distance matrix element, pulling them all into the cache from main memory. The distance matrix occupies a bit less than 8 megabytes and so fits entirely within the level-3 cache, leaving 24 megabytes in the cache to hold other program data, JVM internal data structures, and the program’s machine instructions. Thus, at the end of outer loop iteration 0, all the instructions and all the data the program needs for the three nested loops have been pulled into the fast cache. For the rest of the outer loop iterations, the program runs exclusively out of the cache and does not need to access main memory at all. The same happens with the parallel Floyd’s Algorithm program (FloydSmpRow). Everything fits in the cache, so the program computes results at the same rate regardless of the number of processors, and the program’s efficiency does not change much as $K$ increases.

Next, consider what happens when the sequential program runs with $n$ = 1,260 vertices. Now the distance matrix occupies 12 megabytes and no longer fits entirely in the cache along with the other data and instructions. During one outer loop iteration, as the middle loop pulls different distance matrix rows into the cache, eventually the cache fills up, and the new rows have to replace old rows in the cache. During the next outer loop iteration, the CPU has to reread these replaced rows from main memory, evicting still other rows. Because of this continual **cache churning**, the program computes results more slowly than it would if the whole distance matrix fit in the cache. The same thing happens when the parallel program runs on one processor. But when the parallel program runs on two (or more) processors, the portion of the distance matrix accessed by each thread fits entirely inside the cache again, and the cache churning stops. Thus, the program computes results more quickly, and the program’s efficiency (compared to the sequential version) jumps up abruptly.

As the problem size increases, it takes more processors to make the distance matrix slice accessed by each thread small enough to fit in the cache. Looking at the preceding table, you can see that the efficiency increase always occurs as soon as each thread's distance matrix slice drops below about 8 megabytes. For $n$ = 3,180 vertices, the distance matrix slice never does get small enough to fit entirely in the cache—although it comes close, so there is a slight increase in efficiency for $K$ = 8.

To sum up, a parallel program's speedup need not be caused solely by reducing the number of *calculations* each processor must do. Some of the speedup might also be due to reducing the amount of *data* each processor must access, allowing each processor's data slice to fit in the cache.

## 16.6 Parallel Program with Column Slicing

Earlier, we said that it would also be possible to parallelize Floyd's Algorithm by dividing the distance matrix into column slices. Class FloydSmpCol is an SMP parallel version of Floyd's Algorithm in which each parallel team thread computes all the rows and a subset of the columns of the distance matrix. The source code is identical to class FloydSmpRow, except that in the parallel region, the *innermost* loop is a parallel loop.

```
new ParallelTeam().execute (new ParallelRegion()
   {
   public void run() throws Exception
      {
      for (int ii = 0; ii < n; ++ ii)
         {
         final int i = ii;
         final double[] d_i = d[i];
         for (int r = 0; r < n; ++ r)
            {
            final double[] d_r = d[r];
            execute (0, n-1, new IntegerForLoop()
               {
               public void run (int first, int last)
                  {
                  for (int c = first; c <= last; ++ c)
                     {
                     d_r[c] =
                        Math.min (d_r[c], d_r[i]+d_i[c]);
                     }
                  }
               });
            }
         }
      }
   });
```

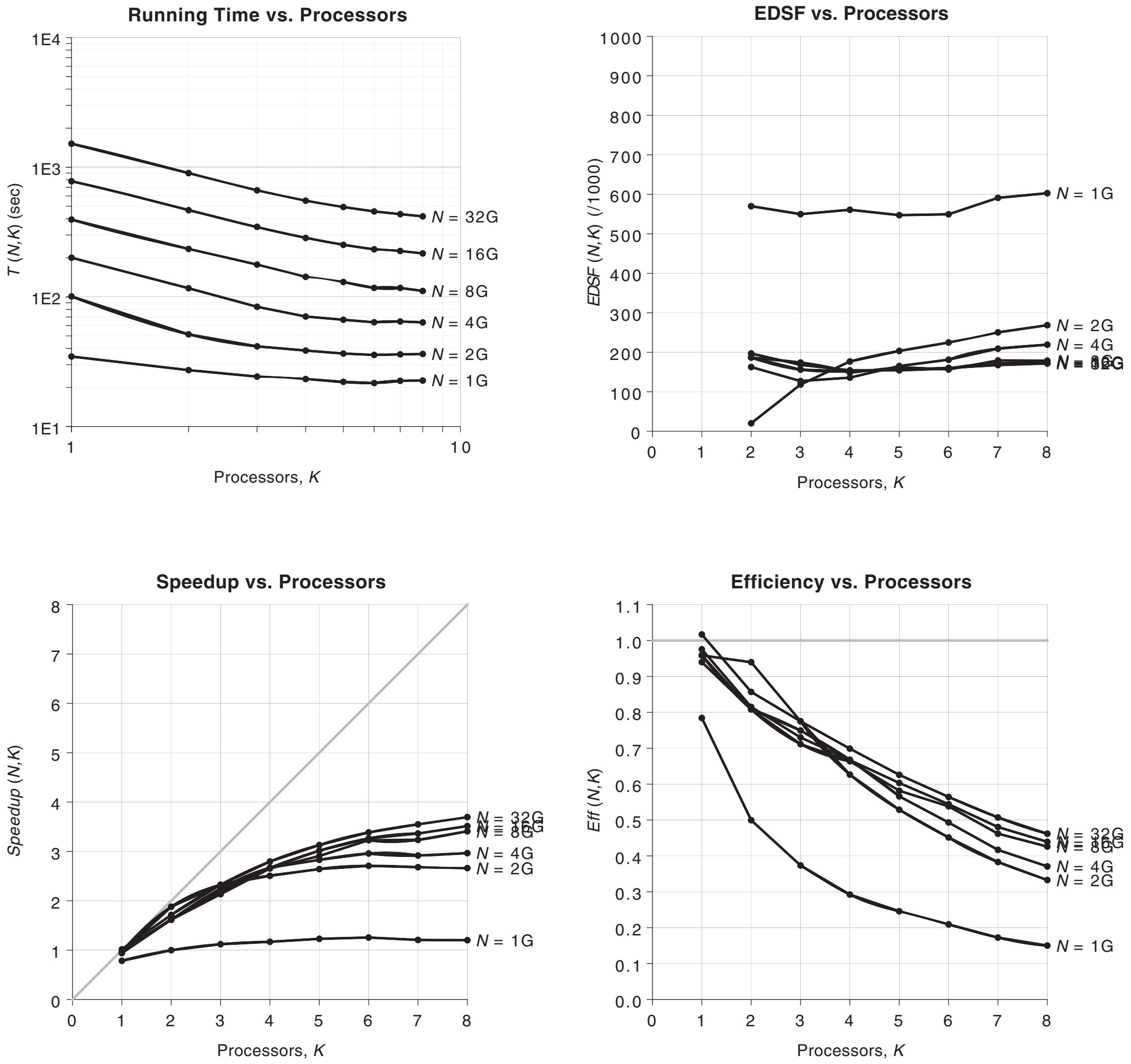
Despite the similarities in the code, this program's performance is completely different. Table 16.2 (at the end of the chapter) lists, and Figure 16.7 plots, the FloydSmpCol program's performance on the "parasite" SMP parallel computer. The running times are for the calculation portion only, not including the distance matrix file I/O. The program was run on the same distance matrices as the FloydSmpRow program.

This time, there are no superlinear speedups and there are no abrupt increases in efficiency. Instead, we see the hallmarks of a parallel program under Amdahl's Law: speedups approaching a limit and efficiencies continually decreasing as *K* increases. The *EDSF* curves show that the FloydSmpCol program suffers from a large sequential fraction: 0.5–0.6 for $N = 1$G ($n = 1{,}000$ vertices), 0.1–0.3 for larger problem sizes. Where is this large sequential fraction coming from? And why didn't we see it in the FloydSmpRow program?

The sequential fraction is caused by the thread synchronization—the implicit barrier wait—at the end of the parallel for loop. The sequential fraction is so large because of the number of barrier waits that take place. In the FloydSmpCol program, there are $n^2$ barrier waits, one at the end of every middle loop iteration. In contrast, the FloydSmpRow program has only *n* barrier waits, one at the end of every outer loop iteration. Compared to the total of $n^3$ computations that take place in Floyd's Algorithm, the sequential fraction due to *n* barrier waits is hardly noticeable, but the sequential fraction due to $n^2$ barrier waits cripples the performance.

Notice that the FloydSmpCol program's sequential fraction decreases as the problem size increases. This is because the total running time *T* is proportional to $n^3$, while the sequential portion of the running time is only proportional to $n^2$. Thus, the sequential fraction *F* is proportional to $n^2/n^3 = n^{-1}$. As *n* increases, *F* decreases; consequently, the speedups and efficiencies improve. This phenomenon is called the **surface-to-volume effect**. (In biology, the surface-to-volume effect refers to how an animal's surface area and volume depend on the animal's length: the surface area is proportional to the square of the length, the volume to the cube of the length.) Whenever a parallel program's sequential running time grows more slowly than the program's total running time, you can get better speedups and efficiencies by scaling up to a larger problem size.

Nonetheless, the moral of the FloydSmpCol program is: *With nested loops, always parallelize the outermost possible loop.* Parallelizing the innermost loop typically yields poor performance.



Running Time vs. Processors

EDSF vs. Processors

Speedup vs. Processors

Efficiency vs. Processors

**Figure 16.7** FloydSeq/FloydSmpCol running-time metrics

## 16.7 For Further Information

Robert Floyd's original publication of the all-pairs shortest-paths algorithm:

- R. Floyd. Algorithm 97: shortest path. *Communications of the ACM*, 5(6):345, June 1962.

Floyd based his algorithm on an algorithm by Stephen Warshall for computing products of Boolean matrices:

- S. Warshall. A theorem on Boolean matrices. *Journal of the ACM*, 9(1):11–12, January 1962.

Other explications of Floyd's Algorithm:

- T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms, Second Edition*. MIT Press, 2001, Section 25.2.
- R. Sedgewick. *Algorithms, Second Edition*. Addison-Wesley, 1988, Chapter 32.
- A. Aho, J. Hopcroft, and J. Ullman. *Data Structures and Algorithms*. Addison-Wesley, 1983, Section 6.4.

**Table 16.1** FloydSeq/FloydSmpRow running-time metrics

| *N* | *K* | *T* | *Spdup* | *Eff* | *EDSF* | *N* | *K* | *T* | *Spdup* | *Eff* | *EDSF* |
|---:|---:|---:|---:|---:|---:|---:|---:|---:|---:|---:|---:|
| 1G | seq | 27191 | | | | 8G | seq | 378557 | | | |
| 1G | 1 | 22726 | 1.196 | 1.196 | | 8G | 1 | 351896 | 1.076 | 1.076 | |
| 1G | 2 | 10847 | 2.507 | 1.253 | -0.045 | 8G | 2 | 195594 | 1.935 | 0.968 | 0.112 |
| 1G | 3 | 8032 | 3.385 | 1.128 | 0.030 | 8G | 3 | 111168 | 3.405 | 1.135 | -0.026 |
| 1G | 4 | 5627 | 4.832 | 1.208 | -0.003 | 8G | 4 | 45701 | 8.283 | 2.071 | -0.160 |
| 1G | 5 | 4736 | 5.741 | 1.148 | 0.010 | 8G | 5 | 40903 | 9.255 | 1.851 | -0.105 |
| 1G | 6 | 4280 | 6.353 | 1.059 | 0.026 | 8G | 6 | 34665 | 10.920 | 1.820 | -0.082 |
| 1G | 7 | 3618 | 7.515 | 1.074 | 0.019 | 8G | 7 | 29432 | 12.862 | 1.837 | -0.069 |
| 1G | 8 | 3246 | 8.377 | 1.047 | 0.020 | 8G | 8 | 24048 | 15.742 | 1.968 | -0.065 |
| 2G | seq | 96580 | | | | 16G | seq | 759572 | | | |
| 2G | 1 | 87071 | 1.109 | 1.109 | | 16G | 1 | 705772 | 1.076 | 1.076 | |
| 2G | 2 | 24056 | 4.015 | 2.007 | -0.447 | 16G | 2 | 393951 | 1.928 | 0.964 | 0.116 |
| 2G | 3 | 15991 | 6.040 | 2.013 | -0.225 | 16G | 3 | 271809 | 2.795 | 0.932 | 0.078 |
| 2G | 4 | 11152 | 8.660 | 2.165 | -0.163 | 16G | 4 | 215268 | 3.528 | 0.882 | 0.073 |
| 2G | 5 | 9318 | 10.365 | 2.073 | -0.116 | 16G | 5 | 134310 | 5.655 | 1.131 | -0.012 |
| 2G | 6 | 7873 | 12.267 | 2.045 | -0.091 | 16G | 6 | 75120 | 10.111 | 1.685 | -0.072 |
| 2G | 7 | 7133 | 13.540 | 1.934 | -0.071 | 16G | 7 | 60768 | 12.500 | 1.786 | -0.066 |
| 2G | 8 | 6552 | 14.741 | 1.843 | -0.057 | 16G | 8 | 49690 | 15.286 | 1.911 | -0.062 |
| 4G | seq | 188427 | | | | 32G | seq | 1542463 | | | |
| 4G | 1 | 175013 | 1.077 | 1.077 | | 32G | 1 | 1444761 | 1.068 | 1.068 | |
| 4G | 2 | 74020 | 2.546 | 1.273 | -0.154 | 32G | 2 | 774616 | 1.991 | 0.996 | 0.072 |
| 4G | 3 | 30180 | 6.243 | 2.081 | -0.241 | 32G | 3 | 520599 | 2.963 | 0.988 | 0.041 |
| 4G | 4 | 24405 | 7.721 | 1.930 | -0.147 | 32G | 4 | 427279 | 3.610 | 0.902 | 0.061 |
| 4G | 5 | 18411 | 10.234 | 2.047 | -0.119 | 32G | 5 | 341923 | 4.511 | 0.902 | 0.046 |
| 4G | 6 | 15753 | 11.961 | 1.994 | -0.092 | 32G | 6 | 285792 | 5.397 | 0.900 | 0.037 |
| 4G | 7 | 15115 | 12.466 | 1.781 | -0.066 | 32G | 7 | 240887 | 6.403 | 0.915 | 0.028 |
| 4G | 8 | 13185 | 14.291 | 1.786 | -0.057 | 32G | 8 | 177570 | 8.687 | 1.086 | -0.002 |

**Table 16.2** FloydSeq/FloydSmpCol running-time metrics

| *N* | *K* | *T* | *Spdup* | *Eff* | *EDSF* | *N* | *K* | *T* | *Spdup* | *Eff* | *EDSF* |
|---:|---:|---:|---:|---:|---:|---:|---:|---:|---:|---:|---:|
| 1G | seq | 27191 | | | | 8G | seq | 378557 | | | |
| 1G | 1 | 34673 | 0.784 | 0.784 | | 8G | 1 | 394757 | 0.959 | 0.959 | |
| 1G | 2 | 27218 | 0.999 | 0.500 | 0.570 | 8G | 2 | 234233 | 1.616 | 0.808 | 0.187 |
| 1G | 3 | 24266 | 1.121 | 0.374 | 0.550 | 8G | 3 | 177376 | 2.134 | 0.711 | 0.174 |
| 1G | 4 | 23254 | 1.169 | 0.292 | 0.561 | 8G | 4 | 142598 | 2.655 | 0.664 | 0.148 |
| 1G | 5 | 22114 | 1.230 | 0.246 | 0.547 | 8G | 5 | 130186 | 2.908 | 0.582 | 0.162 |
| 1G | 6 | 21654 | 1.256 | 0.209 | 0.549 | 8G | 6 | 117264 | 3.228 | 0.538 | 0.156 |
| 1G | 7 | 22520 | 1.207 | 0.172 | 0.591 | 8G | 7 | 117071 | 3.234 | 0.462 | 0.179 |
| 1G | 8 | 22623 | 1.202 | 0.150 | 0.603 | 8G | 8 | 111067 | 3.408 | 0.426 | 0.179 |
| 2G | seq | 96580 | | | | 16G | seq | 759572 | | | |
| 2G | 1 | 100760 | 0.959 | 0.959 | | 16G | 1 | 778678 | 0.975 | 0.975 | |
| 2G | 2 | 51394 | 1.879 | 0.940 | 0.020 | 16G | 2 | 466016 | 1.630 | 0.815 | 0.197 |
| 2G | 3 | 41532 | 2.325 | 0.775 | 0.118 | 16G | 3 | 346826 | 2.190 | 0.730 | 0.168 |
| 2G | 4 | 38542 | 2.506 | 0.626 | 0.177 | 16G | 4 | 284760 | 2.667 | 0.667 | 0.154 |
| 2G | 5 | 36541 | 2.643 | 0.529 | 0.203 | 16G | 5 | 251825 | 3.016 | 0.603 | 0.154 |
| 2G | 6 | 35663 | 2.708 | 0.451 | 0.225 | 16G | 6 | 232688 | 3.264 | 0.544 | 0.159 |
| 2G | 7 | 36019 | 2.681 | 0.383 | 0.250 | 16G | 7 | 225841 | 3.363 | 0.480 | 0.172 |
| 2G | 8 | 36276 | 2.662 | 0.333 | 0.269 | 16G | 8 | 216155 | 3.514 | 0.439 | 0.174 |
| 4G | seq | 188427 | | | | 32G | seq | 1542463 | | | |
| 4G | 1 | 200487 | 0.940 | 0.940 | | 32G | 1 | 1516993 | 1.017 | 1.017 | |
| 4G | 2 | 116544 | 1.617 | 0.808 | 0.163 | 32G | 2 | 900058 | 1.714 | 0.857 | 0.187 |
| 4G | 3 | 83818 | 2.248 | 0.749 | 0.127 | 32G | 3 | 663482 | 2.325 | 0.775 | 0.156 |
| 4G | 4 | 70532 | 2.672 | 0.668 | 0.136 | 32G | 4 | 551848 | 2.795 | 0.699 | 0.152 |
| 4G | 5 | 66588 | 2.830 | 0.566 | 0.165 | 32G | 5 | 492942 | 3.129 | 0.626 | 0.156 |
| 4G | 6 | 63690 | 2.959 | 0.493 | 0.181 | 32G | 6 | 455612 | 3.385 | 0.564 | 0.160 |
| 4G | 7 | 64589 | 2.917 | 0.417 | 0.209 | 32G | 7 | 434572 | 3.549 | 0.507 | 0.168 |
| 4G | 8 | 63534 | 2.966 | 0.371 | 0.219 | 32G | 8 | 417449 | 3.695 | 0.462 | 0.172 |

# CHAPTER 17

# Barrier Actions

in which we learn how to improve a program’s scalability by reducing its memory requirements; we encounter a parallel program that requires sections of sequential code interspersed within the parallel code; and we study the Parallel Java constructs for writing such programs

# CHAPTER 17 Barrier Actions

## 17.1 One-Dimensional Continuous Cellular Automata

A **cellular automaton** (CA) is a simple abstract computing device that is capable of generating all kinds of interesting behavior. For the past 20 years, Stephen Wolfram, the inventor of the symbolic mathematics software package *Mathematica*, has studied CAs and other similar computing devices. In 2002, Wolfram published a monograph, *A New Kind of Science*, describing his work and its implications. We will use Wolfram's **one-dimensional continuous cellular automaton** (1-D CCA) as the subject of our SMP parallel programs in this chapter.

A 1-D CCA consists of a one-dimensional array of **cells** (Figure 17.1). There are $C$ cells in the array, with indexes going from 0 to $C$–1. Each cell has a **value**, a real number in the range 0.0 through 1.0. $X_i$ is the value of the cell at index $i$. This kind of CA is called a *continuous* CA because $X_i$ takes on values from a continuous range. (In another kind of CA, $X_i$ takes on values from a discrete set.)

| | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $X_0$ | $X_1$ | $X_2$ | $X_3$ | $X_4$ | $X_5$ | $X_6$ | $X_7$ | $X_8$ | $X_9$ | $X_{10}$ | $X_{11}$ | $X_{12}$ | $X_{13}$ | $X_{14}$ | $X_{15}$ | $X_{16}$ | $X_{17}$ | $X_{18}$ | $X_{19}$ |

**Figure 17.1** CCA with $C$ = 20 cells

Each cell has a **neighborhood** consisting of the cell itself, the cell to its left, and the cell to its right. The neighborhood of cell $i$ consists of the three cells $\{X_{i-1}, X_i, X_{i+1}\}$. Although we will usually draw the CCA in linear fashion as in Figure 17.1, the cells are actually arranged in a ring (Figure 17.2), and the leftmost cell is adjacent to the rightmost cell. We say the cell array has **wraparound boundaries**. The neighborhood of cell 0 is $\{X_{C-1}, X_0, X_1\}$ as shown in Figure 17.2; the neighborhood of cell $C$–1 is $\{X_{C-2}, X_{C-1}, X_0\}$.

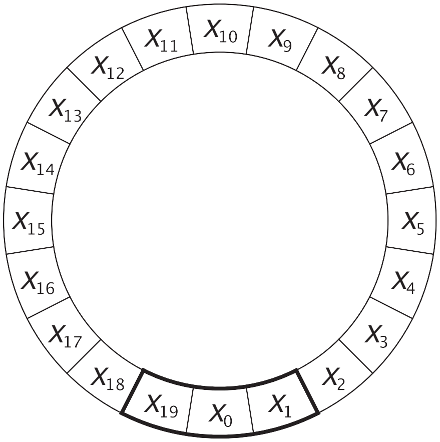


**Figure 17.2** CCA with wraparound boundaries, showing the neighborhood of cell 0

The state of the CCA **evolves** in the following manner. First, the initial state must be specified. While all kinds of initial states are possible, we will use $X_i = 0$ for all *i,* except $X_{C/2} = 1$. That is, all cells are initially 0, except one cell in the middle is initially 1. Then, for each index *i,* a new value is calculated for cell *i* by averaging the current values in cell *i*'s neighborhood, multiplying the average by a constant *A,* adding another constant *B,* and keeping the fractional part of the result (discarding the integer part):

$$X_i^{\text{new}} = \text{frac}\left(\frac{X_{i-1} + X_i + X_{i+1}}{3} \cdot A + B\right) \tag{17.1}$$

Once all the new values have been calculated from the current values, all the cells simultaneously change to their new values. This process repeats as long as desired. As the iterations progress, the CCA evolves through a sequence of states. Different choices for the constants *A* and *B* yield different sequences of states.

Table 17.1 gives an example of the evolution of a CCA with $C = 10$ cells, $A = 1$, and $B = 11/12$ for the first few steps *s* in the state sequence. The cell values are written as exact fractions—rational numbers, or ratios of integers—rather than numbers with a decimal point.

**Table 17.1** Evolution of a CCA—10 cells, 5 steps, $A$=1, $B$=11/12

| *s* | $X_0$ | $X_1$ | $X_2$ | $X_3$ | $X_4$ | $X_5$ | $X_6$ | $X_7$ | $X_8$ | $X_9$ |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 1 | 11/12 | 11/12 | 11/12 | 11/12 | 1/4 | 1/4 | 1/4 | 11/12 | 11/12 | 11/12 |
| 2 | 5/6 | 5/6 | 5/6 | 11/18 | 7/18 | 1/6 | 7/18 | 11/18 | 5/6 | 5/6 |
| 3 | 3/4 | 3/4 | 73/108 | 19/36 | 11/36 | 25/108 | 11/36 | 19/36 | 73/108 | 3/4 |
| 4 | 2/3 | 52/81 | 46/81 | 34/81 | 22/81 | 16/81 | 22/81 | 34/81 | 46/81 | 52/81 |
| 5 | 551/972 | 527/972 | 149/324 | 109/324 | 23/108 | 53/324 | 23/108 | 109/324 | 149/324 | 527/972 |

Instead of listing the cell values numerically, we can depict the cell values graphically (Figure 17.3). Each cell is drawn as a box. The box's color depends on the cell's value: 0 is white; 1 is black; and intermediate values are shades of gray. The cells are laid out horizontally in rows, with the initial state at the top and successive states in the evolution in successive rows, like Table 17.1.

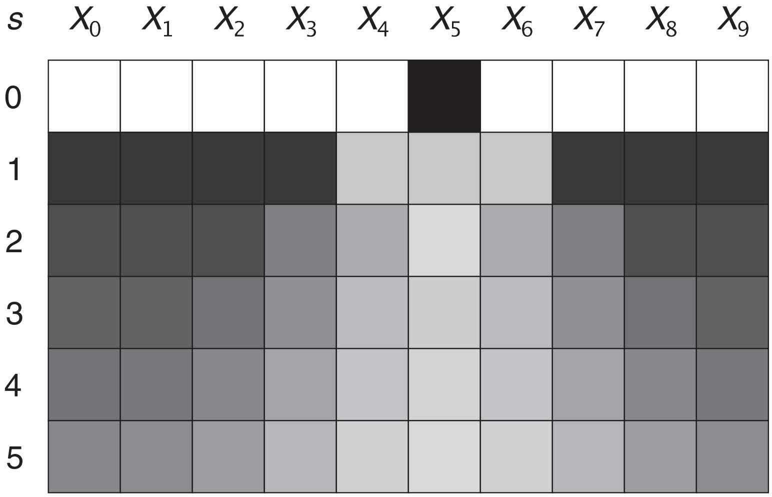


**Figure 17.3** Evolution of a CCA—10 cells, 5 steps, $A$=1, $B$=11/12

Taking this idea to the limit, we can create an image where each pixel represents one cell. Each pixel's gray level represents the corresponding cell's value. The image is $C$ pixels wide by $S$+1 pixels high, where $C$ is the number of cells and $S$ is the number of steps in the CCA evolution. The first row of pixels represents the initial state.

Figure 17.4 shows the evolution of a 400-cell CCA for 200 steps, with $A$ = 1 and $B$ = 11/12. Notice the complex pattern of cell states that emerges. Figure 17.5 shows the evolution of a similar CCA, with a slight change in one parameter: $A$ = 13/12, $B$ = 11/12. Despite having almost identical parameters, the two CCAs' patterns of cell states are completely different. Numerous images of this sort appear in Wolfram's book. Wolfram has studied many kinds of systems, all operating according to simple rules (a CA being one example), but which nonetheless generate complex behavior. Wolfram considers this to be a revolutionary finding, the basis for "a new kind of science" that explains the complexity found in nature better than traditional mathematical models.

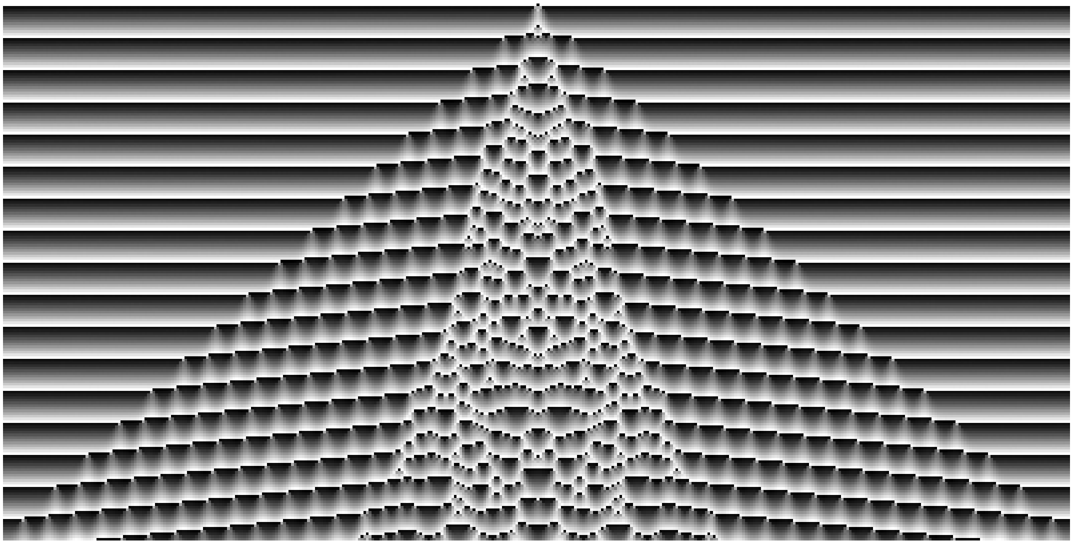


**Figure 17.4:** Evolution of a CCA—400 cells, 200 steps, A=1, B=11⁄12

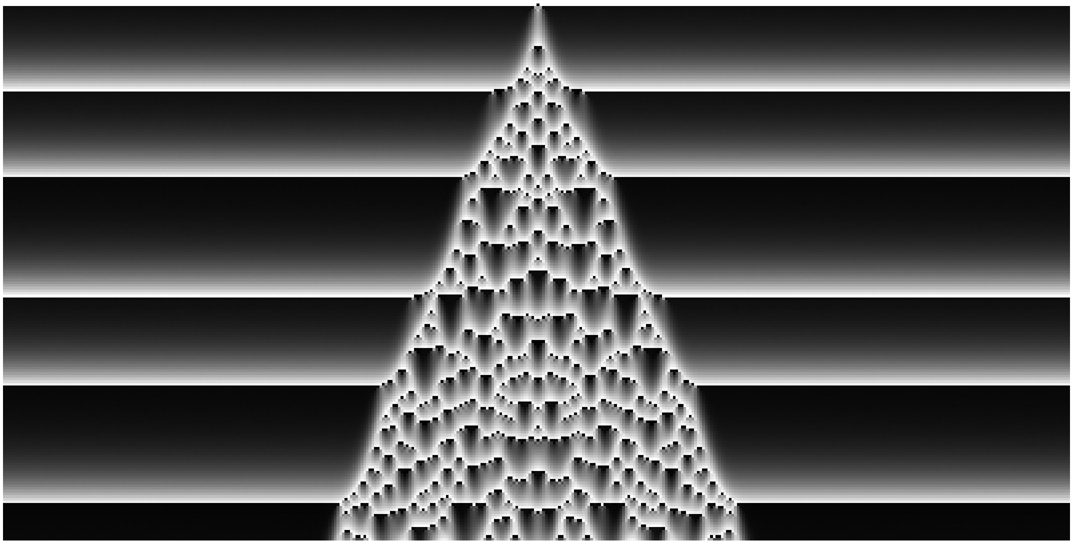


**Figure 17.5:** Evolution of a CCA—400 cells, 200 steps, A=13⁄12, B=11/12

## 17.2 Rational Arithmetic

Earlier, we illustrated the CCA's cell values with exact rational numbers. This was no mere whim. When calculating the states of a CCA, we must use **rational arithmetic** instead of floating-point arithmetic.

Floating-point numbers—even double-precision floating-point numbers—do not have enough precision to calculate the exact cell values. After a while, roundoff errors creep into the calculations, and then the calculated cell states quickly become incorrect. Using exact rational numbers eliminates the roundoff errors.

A **rational number** consists of a **numerator** and a **denominator**, each of which is an integer. The numerator and denominator are stored separately. Recall the formulas for rational arithmetic:

$$\frac{a}{b}+\frac{c}{d}=\frac{ad+bc}{bd} \tag{17.2}$$

$$\frac{a}{b}\times\frac{c}{d}=\frac{ac}{bd} \tag{17.3}$$

$$\text{frac}\left(\frac{a}{b}\right)=\frac{a \text{ rem } b}{b} \tag{17.4}$$

where *a* rem *b* is the remainder when *a* is divided by *b*.

After calculating a formula yielding a rational number, the result should be reduced to lowest terms by dividing any common factors out of the numerator and denominator:

$$\frac{a}{b}=\frac{a \div \text{gcd}(a,b)}{b \div \text{gcd}(a,b)} \tag{17.5}$$

where gcd(*a*,*b*) is the greatest common divisor (GCD) of *a* and *b*. If we don't divide out the GCD, eventually the numerator and denominator become excessively large, leading to slower calculations.

When using rational arithmetic for CCA calculations, the numerator and denominator must be stored as **arbitrary precision integers**—type java.math.BigInteger in Java. Even when we divide out the GCD, the cell values' numerators and denominators quickly become many hundreds of digits long. The Java primitive integer types, `int` and `long`, do not have enough precision to represent the exact cell values.

To determine the gray level for a cell's pixel in the image, the cell's value must be converted from an exact rational number *a/b* to a floating-point number (type `float`) in the range 0 to 1, because the methods for setting a pixel's value in the image take an argument of type `float`. Doing so involves a loss of precision, but that's okay because the image can display only 256 different shades of gray anyway. However, the numerator and the denominator can't be converted to type `float`. The largest value type `float` can represent is $3.40\times10^{38}$, a 39-digit number; this is not large enough to hold a several-hundred-digit number. Even type `double`, which can represent numbers up to $1.80\times10^{308}$, doesn't have a large enough range for some CCAs. Instead, the conversion must be done using **arbitrary precision decimal numbers**—type java.math.BigDecimal in Java.

So as not to clutter the main program with rational arithmetic formulas, a separate class edu.rit.smp.ca.BigRational provides an arbitrary precision rational number. The class has private fields for the numerator and denominator (BigIntegers), and exports these constructors and methods:

- `public BigRational()`—Construct a new rational number whose value is 0.
- `public BigRational (String s)`—Construct a new rational number whose value is parsed from the given string (such as `"11/12"`).

- `public BigRational assign (BigRational x)`—Set this rational number's value to `x`, and return this rational number.
- `public BigRational add (BigRational x)`—Set this rational number's value to the sum of itself and `x`, and return this rational number.
- `public BigRational mul (BigRational x)`—Set this rational number's value to the product of itself and `x`, and return this rational number.
- `public BigRational fracPart()`—Set this rational number's value to the fractional part of itself, and return this rational number.
- `public BigRational normalize()`—Reduce this rational number's numerator and denominator to lowest terms (divide out the GCD), and return this rational number.
- `public float floatValue()`—Return this rational number's value approximated as a single-precision floating-point number.

While this is not a full-featured rational arithmetic class, it suffices for calculating the CCA cell values.

If not for the necessity of using rational arithmetic and arbitrary precision numbers, computing the evolution of a CCA would be very fast. As it is, however, computing even a modestly sized CCA image can require many minutes of calculations and is an attractive candidate for parallel computing.

## 17.3 Improving Memory Scalability

Let's design a program to compute an image of a CCA's evolution. The command-line arguments are the number of cells *C,* the number of steps *S,* and the parameters *A* and *B* in the update formula (17.1). The program generates a grayscale image, like Figures 17.4–17.5, depicting the CCA's evolution and stores the image in a Parallel Java Graphics (PJG) file named on the command line. The program uses class edu.rit.image.PJGGrayImage to create the image. The image's pixel data is stored in a byte matrix (type `byte[][]`).

One way to design the program is to allocate a matrix of BigRational cell values and a pixel data byte matrix, each matrix with *S*+1 rows and *C* columns, one element in each matrix corresponding to one pixel in the image; calculate all the cell and pixel matrix elements; and finally write the entire pixel matrix to the image file.

```
For $c = 0$ to $C–1$:
    $cell_{0,C} \leftarrow 0$
    $pixel_{0,C} \leftarrow 0$
$cell_{0,C/2} \leftarrow 1$
$pixel_{0,C/2} \leftarrow 1$
For $s = 1$ to $S$:
    For c = 0 to $C–1$:
        $cell_{s,c} \leftarrow \mathrm{frac}((cell_{s-1,c-1} + cell_{s-1,c} + cell_{s-1,c+1}) \times 1/3 \times A + B)$
        $pixel_{s,c} \leftarrow \mathrm{floatValue}(cell_{s,c})$
Write *pixel* data to PJG image file
```

However, there's a problem with this design: it requires $O(SC)$ memory to hold all the data. This means that when scaling up the problem on an SMP parallel computer, the size of the physical memory limits both the number of cells *and* the number of steps we can compute.

To improve this program's scalability, we must reduce its memory requirements. To calculate a cell value in a certain row, we need the cell values in the previous row only, not in all prior rows. This means we don't need a whole matrix of cell values; rather, we need only two arrays, the current cell values and the next cell values. We can't update the cell values in place because we need each current cell value in the calculation of three next cell values. To advance to the next time step after calculating all the next cell values, we merely swap the two array references; this is faster than copying the next cell array back to the current cell array. Likewise, we don't need a whole matrix of pixel data values, but only an array to hold the current row of pixels. The new design follows:

```
For $c = 0$ to $C–1$:
    $currCell_c \leftarrow 0$
$currCell_{C/2} \leftarrow 1$
For $s = 0$ to $S–1$:
    For $c = 0$ to $C–1$:
        $nextCell_c \leftarrow \text{frac}((currCell_{c-1} + currCell_c + currCell_{c+1}) \times 1/3 \times A + B)$
    For $c = 0$ to $C–1$:
        $pixel_c \leftarrow \text{floatValue}(currCell_c)$
    Write $pixel$ data as row $s$ of PJG image file
    Swap ($currCell$, $nextCell$)
For $c = 0$ to $C–1$:
    $pixel_c \leftarrow \text{floatValue}(currCell_c)$
Write $pixel$ data as row $S$ of PJG image file
```

Note that if we are going to keep an array with only one row of pixels, we need to write the PJG image file one row at a time, not all at once. (The PJG image classes support reading and writing PJG files in multiple separate chunks.) Now the program requires only $O(C)$ storage, not $O(SC)$. As we scale up the problem, the physical memory limits only the number of cells we can compute, not the number of steps. Of course, the size of the disk drive on which we are writing the image file still limits the number of steps we can do; but disk drives are orders of magnitude larger than CPU main memories.

## 17.4 Sequential Program

Here is the source code for class edu.rit.smp.ca.CCASeq, the sequential version of the program to calculate an image of a CCA's evolution.

```
package edu.rit.smp.ca;
import edu.rit.image.GrayImageRow;
import edu.rit.image.PJGGrayImage;
import edu.rit.image.PJGImage;
import edu.rit.util.Range;
import java.io.BufferedOutputStream;
import java.io.File;
import java.io.FileOutputStream;
import java.io.IOException;
public class CCASeq
   {
   // Constants.
   static final BigRational ZERO = new BigRational ("0");
   static final BigRational ONE = new BigRational ("1");
   static final BigRational ONE_THIRD = new BigRational ("1/3");

   // Command line arguments.
   static int C;
   static int S;
   static BigRational A;
   static BigRational B;
   static File imagefile;

   // Old and new cell arrays.
   static BigRational[] currentCell;
   static BigRational[] nextCell;
```

We will use class edu.rit.image.PJGGrayImage to write the PJG image file. Class PJGGrayImage requires the pixel data to be stored in a byte matrix, namely the variable `pixelmatrix`. However, we will not allocate separate storage for each row of the matrix. Instead, we will allocate storage for only one row, namely the variable `pixelrow`. The variable `imagerow`, an instance of class edu.rit.image.GrayImageRow, lets us manipulate the pixels in one row of an image, which is stored in a byte array.

```
   // Grayscale image matrix.
   static byte[][] pixelmatrix;
   static PJGGrayImage image;
   static PJGImage.Writer writer;

   // One row of the grayscale image matrix.
   static byte[] pixelrow;
   static GrayImageRow imagerow;

   /**
    * Main program.
    */
   public static void main
      (String[] args)
      throws Exception
      {
      // Start timing.
      long t1 = System.currentTimeMillis();

      // Parse command line arguments.
      if (args.length != 5) usage();
      C = Integer.parseInt (args[0]);
      S = Integer.parseInt (args[1]);
```

Because the factor *A*/3 in the update formula (17.1) is a constant, we calculate it once at this point rather than repeatedly in the cell update loop.

```
      A = new BigRational (args[2]) .mul (ONE_THIRD);
      B = new BigRational (args[3]);
      imagefile = new File (args[4]);

      // Allocate storage for old and new cell arrays. Initialize
      // all cells to 0, except center cell to 1.
      currentCell = new BigRational [C];
      nextCell = new BigRational [C];
      for (int i = 0; i < C; ++ i)
         {
         currentCell[i] = new BigRational();
         nextCell[i] = new BigRational();
         }
      currentCell[C/2].assign (ONE);

      // Set up pixel matrix, image, and image writer.
      pixelmatrix = new byte [S+1] [];
      image = new PJGGrayImage (S+1, C, pixelmatrix);
      writer =
         image.prepareToWrite
            (new BufferedOutputStream
               (new FileOutputStream (imagefile)));

      // Allocate storage for one pixel matrix row.
      pixelrow = new byte [C];
      imagerow = new GrayImageRow (pixelrow);
```

When storing a pixel value, the default is to interpret 0 as black and 1 as white. We want it the other way around: 0 as white and 1 as black.

```
      imagerow.setInterpretation (PJGGrayImage.ZERO_IS_WHITE);

      // Do S time steps.
      for (int s = 0; s < S; ++ s)
         {
         // Calculate next state of each cell.
         for (int i = 0; i < C; ++ i)
            {
```

Here is where we calculate the update formula (17.1). The index expression `(i-1+C)%C` refers to the left neighbor of cell *i,* wrapping around to index *C*–1 if *i* is 0. The index expression `(i+1)%C` refers to the right neighbor of cell *i,* wrapping around to index 0 if *i* is *C*–1. Remember that *A* was premultiplied by 1⁄3. Normalizing the result once at the end of the formula results in a smaller running time than normalizing after each step of the formula.

```
         nextCell[i]
            .assign (currentCell[i])
            .add (currentCell[(i-1+C)%C])
            .add (currentCell[(i+1)%C])
            .mul (A)
            .add (B)
            .normalize()
            .fracPart();
         }
      // Write current CA state to image file.
      writeCurrentCell (s);

      // Advance one time step — swap old and new cell arrays.
      BigRational[] tmp = currentCell;
      currentCell = nextCell;
      nextCell = tmp;
      }

   // Write final CA state to image file.
   writeCurrentCell (S);
   writer.close();

   // Stop timing.
   long t2 = System.currentTimeMillis();
   System.out.println ((t2-t1) + " msec total");
   }
```

We put this bit of pseudocode:

```
For $c = 0$ to $C–1$:
    $pixel_c$ ← floatValue($currCell_c$)
Write $pixel$ data as row $s$ of PJG image file
```

in a subroutine, `writeCurrentCell()`, because it appears in two places in the algorithm.

```
/**
 * Write the current cell array to the given row of the image
 * file.
 */
private static void writeCurrentCell
   (int r)
   throws IOException
   {
   // Set image row's gray values based on current cell states.
   for (int i = 0; i < C; ++ i)
      {
      imagerow.setPixel (i, currentCell[i].floatValue());
      }
```

Here is where we set each row of the pixel matrix to refer to the already-allocated pixel array, which we have just filled with the current row's pixel data.

```
   // Set row r of the pixel matrix.
   pixelmatrix[r] = pixelrow;
```

And here is where we write one chunk of pixel data to the PJG image file. The `writeRowSlice()` method writes the rows from the lower bound to the upper bound, inclusive, of the given range. The pixel data comes from row `r` of the `pixelmatrix` variable.

```
   // Write row-r slice of the image to the image file.
   writer.writeRowSlice (new Range (r, r));
   }
}
```

## 17.5 Barrier Actions

To design a parallel version of this program, we must first analyze the algorithm for sequential dependencies. There is a sequential dependency between steps in the evolution; we cannot compute the cell values for the next step until we finish computing the cell values for the current step. Therefore, the outer loop must remain a sequential loop. However, the value calculated for any element in the *nextCell* array does not depend on the value of any other element in the *nextCell* array; there are no sequential dependencies in the inner loop. Therefore, the inner loop can become a parallel loop, as follows:

```
For $c = 0$ to $C–1$:
    $currCell_c \leftarrow 0$
$currCell_{C/2} \leftarrow 1$
For $s = 0$ to $S–1$:
    Parallel for $c = 0$ to $C–1$:
        $nextCell_c \leftarrow \mathrm{frac}((currCell_{c-1} + currCell_c + currCell_{c+1}) \times 1/3 \times A + B)$
    Compute and write pixel data as row $s$ of PJG image file
    Swap ($currCell$, $nextCell$)
Compute and write pixel data as row $S$ of PJG image file
```

As we did with the program for Floyd's Algorithm in Chapter 16, we enclose both the outer and the inner loops in a parallel region of code executed by a parallel team of threads, as follows:

```
For $c$ = 0 to $C$–1:
    $currCell_c \leftarrow 0$
$currCell_{C/2} \leftarrow 1$
Parallel region:
    For $s$ = 0 to $S$–1:
        Parallel for $c$ = 0 to $C$–1:
        $nextCell_c \leftarrow \mathrm{frac}((currCell_{c-1} + currCell_c + currCell_{c+1}) \leftarrow 1/3 \times A + B)$
    Compute and write pixel data as row $s$ of PJG image file
    Swap ($currCell$, $nextCell$)
Compute and write pixel data as row $S$ of PJG image file
```

As was the case for the parallel Floyd's Algorithm program, the implicit barrier wait at the end of the parallel for loop is crucial. The threads cannot be allowed to proceed to the next step of the evolution until all threads have finished the current step. But this time, there's a twist. Before going to the next outer loop iteration, some processing has to happen, namely computing and writing the pixel row to the image file and swapping the cell array references. Furthermore, this processing must be done by *only one thread,* not by all the threads that are executing the parallel region. Having each parallel team thread compute and write the pixel row would store incorrect data in the image file; besides, class PJGGrayImage is not multiple-thread safe. Having each parallel team thread swap the cell array references would likewise result in chaos.

Parallel Java provides the **barrier action** for situations like this. When executing a parallel for loop, you can include an instance of class edu.rit.pj.BarrierAction (Figure 17.6).

```
new ParallelTeam().execute (new ParallelRegion()
   {
   . . .
   execute (0, 99, new IntegerForLoop()
      {
      public void run (int first, int last)
         {
         for (int i = first; i <= last; ++ i)
            {
            // Loop body code goes here
            }
         }
      },
   new BarrierAction()
```

```
            {
            public void run()
               {
               // Code to be executed in a single thread goes here
               }
            });
         . . .
         });
```

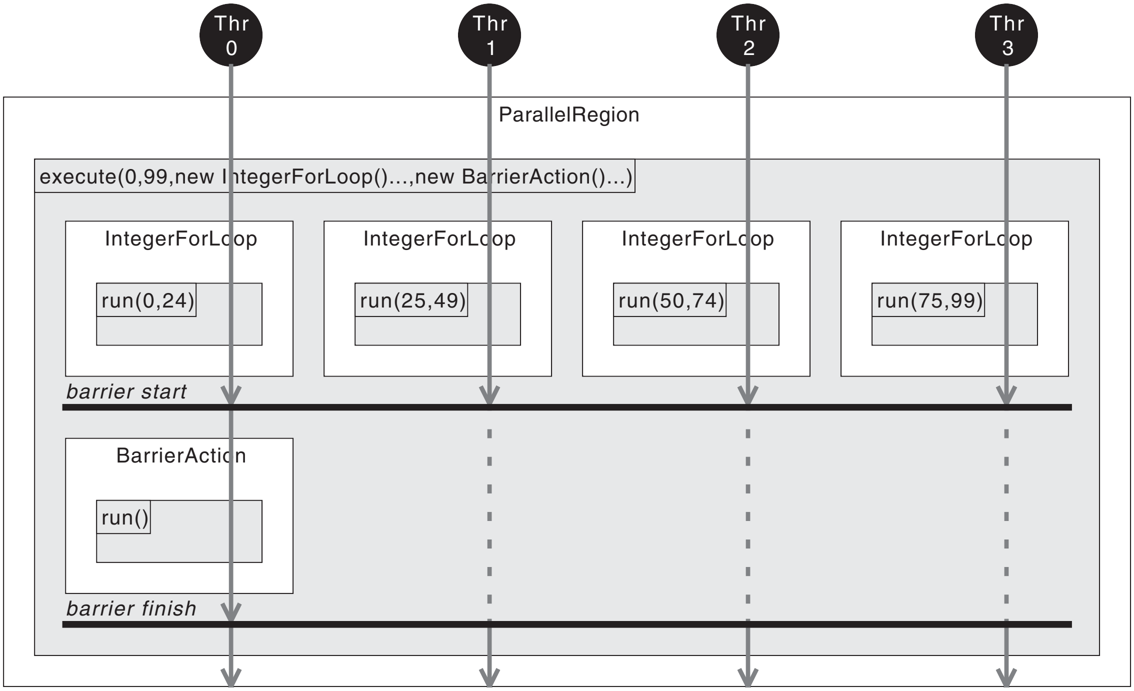


**Figure 17.6** A parallel for loop with a barrier action

After finishing the parallel for loop, each parallel team thread waits at the barrier. When all threads have arrived at the barrier, one of the threads calls the barrier action's `run()` method; during this time, the other threads continue to wait at the barrier. When the barrier action's `run()` method returns, all threads resume, leave the barrier, and execute whatever comes after the parallel for loop in the parallel region.

Instead of specifying a barrier action object, you can specify the constant `BarrierAction.WAIT`.

```
         execute (0, 99, new IntegerForLoop()
            {
            . . .
```

```
      },
   BarrierAction.WAIT);
```

In this case, after finishing the parallel for loop, each parallel team thread waits at the barrier. When all threads have arrived at the barrier, no single-threaded barrier action is executed. All threads simply leave the barrier and execute whatever comes after the parallel for loop in the parallel region. (This is the default behavior if the barrier action is omitted.)

You can also specify the constant `BarrierAction.NO_WAIT`.

```
   execute (0, 99, new IntegerForLoop()
      {
      . . .
      },
   BarrierAction.NO_WAIT);
```

Specifying `NO_WAIT` eliminates the barrier. After finishing the parallel for loop, each parallel team thread does *not* wait for the others, but immediately proceeds to execute whatever comes after the parallel for loop in the parallel region. (There is no single-threaded barrier action, either.)

## 17.6 Parallel Program

Pulling it all together, the parallel CCA program consists of a sequential outer loop and a parallel inner loop with a barrier action.

```
For $c$ = 0 to $C$–1:
    $currCell_c \leftarrow 0$
$currCell_{C/2} \leftarrow 1$
Parallel region:
    For $s$ = 0 to $S$–1:
        Parallel for $c$ = 0 to $C$–1: ($K$ threads)
            $nextCell_c \leftarrow \mathrm{frac}((currCell_{c-1} + currCell_c + currCell_{c+1}) \times 1/3 \times A + B)$
        Barrier action: (single thread)
            Compute and write $pixel$ data as row $s$ of PJG image file
            Swap ($currCell$, $nextCell$)
Compute and write $pixel$ data as row $S$ of PJG image file
```

To finish the design, we must decide which variables will be shared and whether to synchronize multiple threads accessing the shared variables. The command-line arguments are shared WORM variables that need no synchronization. The *currCell* array is shared, but during the parallel for loop the threads only read it; therefore, it needs no synchronization. The *nextCell* array is shared, but during the parallel for loop each array element is written by only one thread; therefore, it needs no synchronization. The code to compute the pixel data array, write it to the image file, and swap the cell array references is synchronized by the barrier action.

Here is the source code for class edu.rit.smp.ca.CCASmp, the parallel version of the program.

```
package edu.rit.smp.ca;
import edu.rit.image.GrayImageRow;
import edu.rit.image.PJGGrayImage;
import edu.rit.image.PJGImage;
import edu.rit.pj.BarrierAction;
import edu.rit.pj.IntegerForLoop;
import edu.rit.pj.IntegerSchedule;
import edu.rit.pj.ParallelRegion;
import edu.rit.pj.ParallelTeam;
import edu.rit.util.Range;
import java.io.BufferedOutputStream;
import java.io.File;
import java.io.FileOutputStream;
import java.io.IOException;
public class CCASmp
   {
   // Constants.
   static final BigRational ZERO = new BigRational ("0");
   static final BigRational ONE = new BigRational ("1");
   static final BigRational ONE_THIRD = new BigRational ("1/3");

   // Command line arguments.
   static int C;
   static int S;
   static BigRational A;
   static BigRational B;
   static File imagefile;

   // Old and new cell arrays.
   static BigRational[] currentCell;
   static BigRational[] nextCell;

   // Grayscale image matrix.
   static byte[][] pixelmatrix;
   static PJGGrayImage image;
   static PJGImage.Writer writer;

   // One row of the grayscale image matrix.
   static byte[] pixelrow;
   static GrayImageRow imagerow;

   /**
    * Main program.
    */
```

```
public static void main
   (String[] args)
   throws Exception
   {
   // Start timing.
   long t1 = System.currentTimeMillis();

   // Parse command line arguments.
   if (args.length != 5) usage();
   int argi = 0;
   C = Integer.parseInt (args[0]);
   S = Integer.parseInt (args[1]);
   A = new BigRational (args[2]) .mul (ONE_THIRD);
   B = new BigRational (args[3]);
   imagefile = new File (args[4]);

   // Allocate storage for old and new cell arrays. Initialize
   // all cells to 0, except center cell to 1.
   currentCell = new BigRational [C];
   nextCell = new BigRational [C];
   for (int i = 0; i < C; ++ i)
      {
      currentCell[i] = new BigRational();
      nextCell[i] = new BigRational();
      }
   currentCell[C/2].assign (ONE);

   // Set up pixel matrix, image, and image writer.
   pixelmatrix = new byte [S+1] [];
   image = new PJGGrayImage (S+1, C, pixelmatrix);
   writer =
      image.prepareToWrite
         (new BufferedOutputStream
            (new FileOutputStream (imagefile)));

   // Allocate storage for one pixel matrix row.
   pixelrow = new byte [C];
   imagerow = new GrayImageRow (pixelrow);
   imagerow.setInterpretation (PJGGrayImage.ZERO_IS_WHITE);

   // Parallel calculation section.
   new ParallelTeam().execute (new ParallelRegion()
      {
      public void run() throws Exception
         {
         // Do S time steps. Sequential outer loop.
         for (int s = 0; s < S; ++ s)
            {
            final int step = s;
```

Here is the inner parallel for loop over the cells in the cell array. Dividing the cells evenly among the parallel team threads usually results in an unbalanced load. In Figure 17.4, for example, the central cell values have many more digits, and require more time to calculate, than the outer cells. To balance the load, we hard-code a guided schedule into the parallel for loop.

```
                  // Calculate next state of each cell. Parallel inner
                  // loop with a barrier action.
                  execute (0, C-1, new IntegerForLoop()
                     {
                     public IntegerSchedule schedule()
                        {
                        return IntegerSchedule.guided();
                        }
                     public void run (int first, int last)
                        {
                        for (int i = first; i <= last; ++ i)
                           {
                           nextCell[i]
                              .assign (currentCell[i])
                              .add (currentCell[(i-1+C)%C])
                              .add (currentCell[(i+1)%C])
                              .mul (A)
                              .add (B)
                              .normalize()
                              .fracPart();
                           }
                        }
                     },
```

And here is the barrier action.

```
                  // Synchronize threads before next outer loop
                  // iteration.
                  new BarrierAction()
                     {
                     public void run() throws Exception
                        {
                        // Write current CA state to image file.
                        writeCurrentCell (step);
                        // Advance one time step — swap old and new
                        // cell arrays.
                        BigRational[] tmp = currentCell;
                        currentCell = nextCell;
                        nextCell = tmp;
                        }
                     });
                  }
               }
            });

         // Write final CA state to image file.
         writeCurrentCell (S);
         writer.close();

         // Stop timing.
         long t2 = System.currentTimeMillis();
         System.out.println ((t2-t1) + " msec total");
         }

      /**
       * Write the current cell array to the given row of the image
       * file.
       */
      private static void writeCurrentCell
         (int r)
         throws IOException
         {
         // Set image row’s gray values based on current cell states.
         for (int i = 0; i < C; ++ i)
            {
            imagerow.setPixel (i, currentCell[i].floatValue());
            }

         // Set row r of the pixel matrix.
         pixelmatrix[r] = pixelrow;

         // Write row-r slice of the image to the image file.
         writer.writeRowSlice (new Range (r, r));
         }
      }
```

Table 17.2 (at the end of the chapter) lists, and Figure 17.7 plots, the CCASmp program's performance on the "parasite" SMP parallel computer. The command line arguments were $C = 2{,}000$ cells; $S = 400$, 500, 600, 700, 800, and 1,000 steps; $A = 1$; and $B = 11/12$. The *EDSF* curves evince a rather large sequential fraction, resulting in less-than-ideal speedups and efficiencies. Might there be a way to improve this parallel program's performance? That will be the topic of Chapter 18.

## 17.7 For Further Information

On cellular automata:

- S. Wolfram. *A New Kind of Science*. Stephen Wolfram, LLC, 2002.
- A New Kind of Science Online. http://www.wolframscience.com/

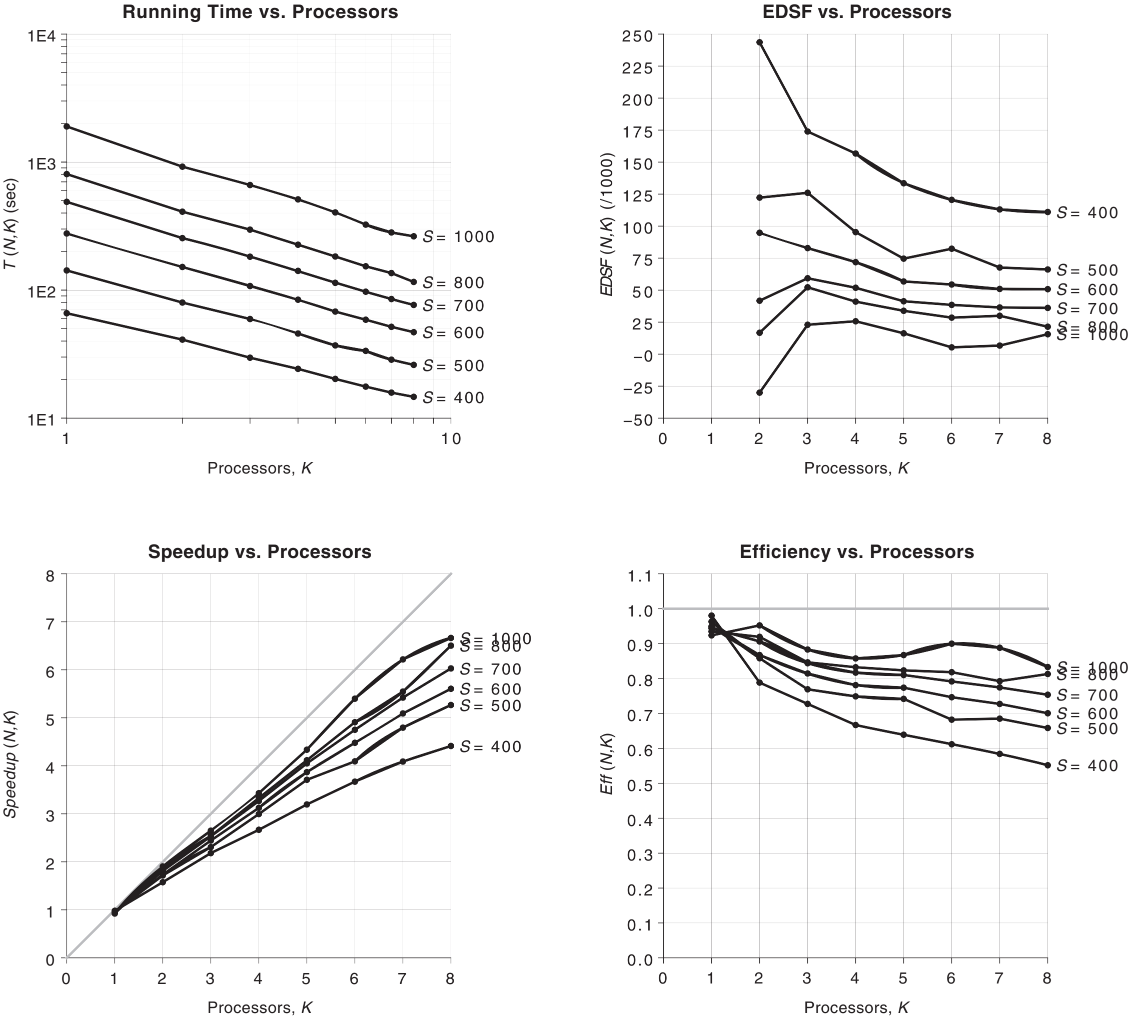


**Figure 17.7** CCASeq/CCASmp running-time metrics

**Table 17.2** CCASeq/CCASmp running-time metrics

| *S* | *K* | *T* | *Spdup* | *Eff* | *EDSF* | *S* | *K* | *T* | *Spdup* | *Eff* | *EDSF* |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 400 | seq | 64678 | | | | 700 | seq | 461504 | | | |
| 400 | 1 | 65972 | 0.980 | 0.980 | | 700 | 1 | 488912 | 0.944 | 0.944 | |
| 400 | 2 | 41021 | 1.577 | 0.788 | 0.244 | 700 | 2 | 254648 | 1.812 | 0.906 | 0.042 |
| 400 | 3 | 29638 | 2.182 | 0.727 | 0.174 | 700 | 3 | 182282 | 2.532 | 0.844 | 0.059 |
| 400 | 4 | 24245 | 2.668 | 0.667 | 0.157 | 700 | 4 | 141228 | 3.268 | 0.817 | 0.052 |
| 400 | 5 | 20239 | 3.196 | 0.639 | 0.133 | 700 | 5 | 113927 | 4.051 | 0.810 | 0.041 |
| 400 | 6 | 17619 | 3.671 | 0.612 | 0.120 | 700 | 6 | 97148 | 4.751 | 0.792 | 0.038 |
| 400 | 7 | 15815 | 4.090 | 0.584 | 0.113 | 700 | 7 | 85119 | 5.422 | 0.775 | 0.036 |
| 400 | 8 | 14654 | 4.414 | 0.552 | 0.111 | 700 | 8 | 76548 | 6.029 | 0.754 | 0.036 |
| 500 | seq | 137161 | | | | 800 | seq | 753220 | | | |
| 500 | 1 | 142423 | 0.963 | 0.963 | | 800 | 1 | 805800 | 0.935 | 0.935 | |
| 500 | 2 | 79915 | 1.716 | 0.858 | 0.122 | 800 | 2 | 409614 | 1.839 | 0.919 | 0.017 |
| 500 | 3 | 59437 | 2.308 | 0.769 | 0.126 | 800 | 3 | 296633 | 2.539 | 0.846 | 0.052 |
| 500 | 4 | 45783 | 2.996 | 0.749 | 0.095 | 800 | 4 | 226226 | 3.330 | 0.832 | 0.041 |
| 500 | 5 | 36980 | 3.709 | 0.742 | 0.075 | 800 | 5 | 182961 | 4.117 | 0.823 | 0.034 |
| 500 | 6 | 33507 | 4.094 | 0.682 | 0.082 | 800 | 6 | 153422 | 4.909 | 0.818 | 0.028 |
| 500 | 7 | 28600 | 4.796 | 0.685 | 0.068 | 800 | 7 | 135786 | 5.547 | 0.792 | 0.030 |
| 500 | 8 | 26041 | 5.267 | 0.658 | 0.066 | 800 | 8 | 115802 | 6.504 | 0.813 | 0.021 |
| 600 | seq | 262908 | | | | 1000 | seq | 1753323 | | | |
| 600 | 1 | 276906 | 0.949 | 0.949 | | 1000 | 1 | 1898334 | 0.924 | 0.924 | |
| 600 | 2 | 151574 | 1.735 | 0.867 | 0.095 | 1000 | 2 | 920580 | 1.905 | 0.952 | -0.030 |
| 600 | 3 | 107588 | 2.444 | 0.815 | 0.083 | 1000 | 3 | 661783 | 2.649 | 0.883 | 0.023 |
| 600 | 4 | 84142 | 3.125 | 0.781 | 0.072 | 1000 | 4 | 511109 | 3.430 | 0.858 | 0.026 |
| 600 | 5 | 67950 | 3.869 | 0.774 | 0.057 | 1000 | 5 | 404329 | 4.336 | 0.867 | 0.016 |
| 600 | 6 | 58693 | 4.479 | 0.747 | 0.054 | 1000 | 6 | 324731 | 5.399 | 0.900 | 0.005 |
| 600 | 7 | 51639 | 5.091 | 0.727 | 0.051 | 1000 | 7 | 282019 | 6.217 | 0.888 | 0.007 |
| 600 | 8 | 46905 | 5.605 | 0.701 | 0.051 | 1000 | 8 | 263116 | 6.664 | 0.833 | 0.016 |

*This page intentionally left blank*

# CHAPTER 18 Overlapping

in which we learn why input/output can reduce a parallel program's performance; we discover how doing the computation and I/O in parallel can restore the program's performance; and we learn how to write parallel programs with overlapped computation and I/O

# CHAPTER 18 Overlapping

## 18.1 Overlapped Computation and I/O

The one-dimensional continuous cellular automaton program in Chapter 17 suffered from a rather large sequential fraction, leading to poor speedups and efficiencies. It's fairly obvious what's causing the large sequential fraction. We coded it ourselves. It is the barrier action that computes and writes the pixel row to the image file. By its nature, file I/O must be done sequentially, and that is why we put this code in the single-threaded barrier action. The time spent executing the statements in the barrier action, along with the time required to synchronize the threads before and after the barrier action, is time the program is not running in parallel.

However, in this program, nothing requires us to finish calculating all the cells' *next* states before starting to write the cells' *current* states to the image file. Because the current states do not change during the next-state computations, we can start the next round of computations and the I/O at the same time. In fact, why not do the computations in one team of threads and the I/O in a separate thread? (The threads have to synchronize with each other so that the I/O thread doesn't get ahead of the computation threads.) Better yet, why not take advantage of the SMP parallel computer's hardware capabilities and run the I/O thread on its own processor, in parallel with the computation threads running on the other processors? This technique is called **overlapped computation and I/O**, or just **overlapping**.

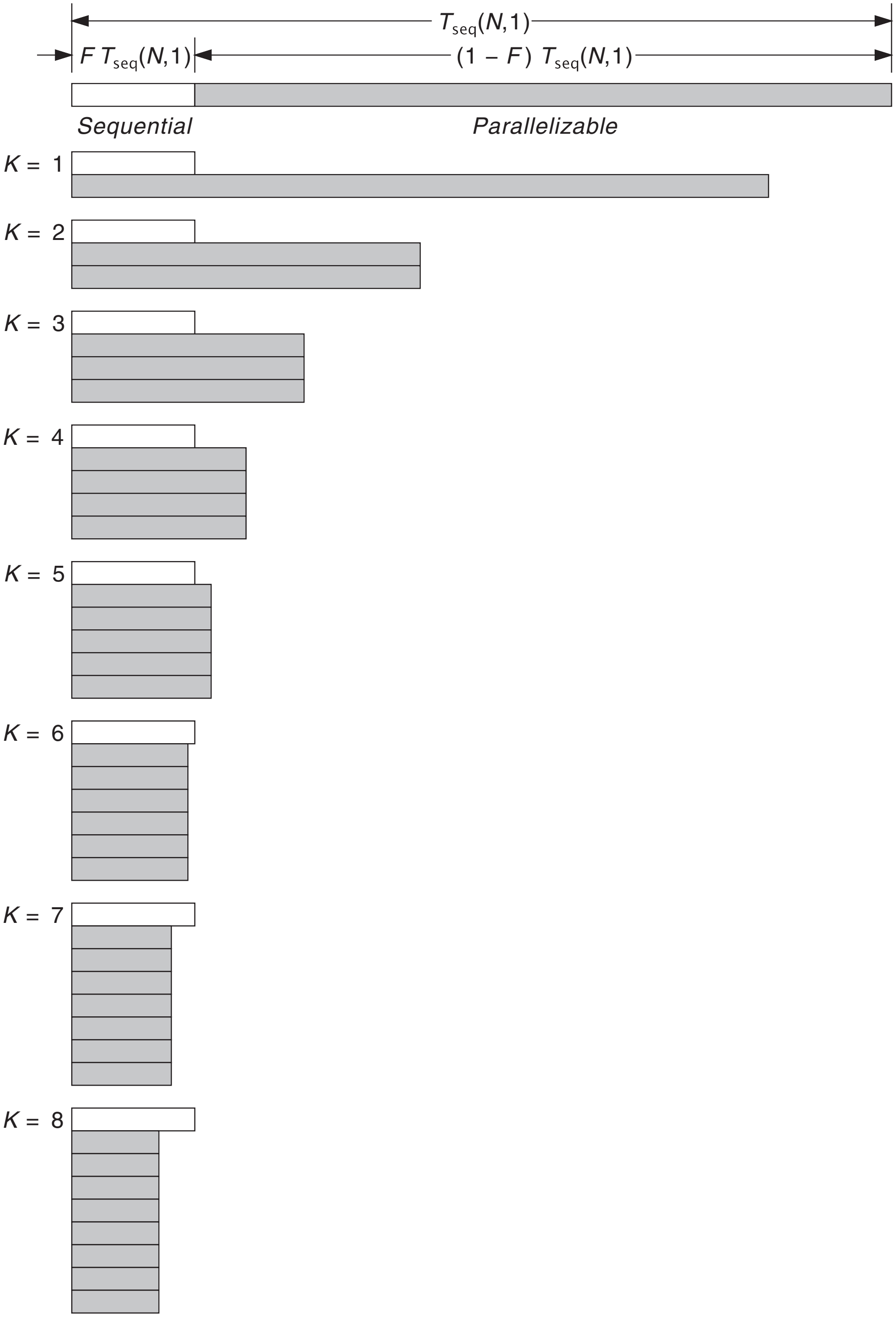


**Figure 18.1** A parallel program with running time $T_{seq}(N,1)$ and sequential fraction $F$ using overlapping with different numbers of processors $K$

Figure 18.1 shows what happens in the ideal situation where all the program's sequential portion can be overlapped with the parallel portion. The program's running time is the *maximum* of the sequential portion's running time and the parallel portion's running time, rather than the sum. The running time is

$$T_{\text{par}}(N,K) = \max\left( F \cdot T_{\text{seq}}(N,1),\ \frac{1}{K}(1-F) \cdot T_{\text{seq}}(N,1) \right) \tag{18.1}$$

where $T_{\text{seq}}(N,1)$ is the running time for the sequential version of the program, $F$ is the sequential fraction, $K$ is the number of processors doing the calculation section in parallel (not counting the extra processor doing the I/O), and $T_{\text{par}}(N,K)$ is the running time for the parallel version of the program executing on $K$+1 processors.

From Equation 18.1, we can derive equations for speedup and efficiency with overlapping:

$$\begin{aligned} \mathit{Speedup}(N,K) &= \frac{T_{\text{seq}}(N,1)}{T_{\text{par}}(N,K)} \\ &= \frac{T_{\text{seq}}(N,1)}{\max\left( F \cdot T_{\text{seq}}(N,1),\ \frac{1}{K}(1-F) \cdot T_{\text{seq}}(N,1) \right)} \\ &= \min\left( \frac{1}{F},\ \frac{K}{1-F} \right) \end{aligned} \tag{18.2}$$

$$\mathit{Eff}(N,K) = \frac{\mathit{Speedup}(N,K)}{K} = \min\left( \frac{1}{KF},\ \frac{1}{1-F} \right) \tag{18.3}$$

As $K$ increases, *Speedup* goes to a limit of $1/F$ and *Eff* goes to a limit of 0. These are the same limits Amdahl's Law predicts for a program that doesn't use overlapping. So what difference does overlapping make?

The difference becomes clear when we plot the speedup both with and without overlapping (Figure 18.2). Without overlapping, the speedup approaches its limit gradually as $K$ increases, and the speedups are far from linear even with a modest number of processors. With overlapping, the speedup is linear right up until it hits its limit, and then the speedup remains constant. In fact, with overlapping the speedup is **superlinear**. The additional processor doing the I/O (processor $K$+1) causes $T_{\text{par}}(N,K)$ to be less than $T_{\text{seq}}(N,1)/K$, hence the speedup is greater than $K$.

Figure 18.3 shows the expected shapes of the efficiency curves with and without overlapping. With overlapping, the efficiency stays constant until the speedup hits its limit, and then the efficiency decreases in proportion to $1/K$. This results in higher efficiencies than a program without overlapping.

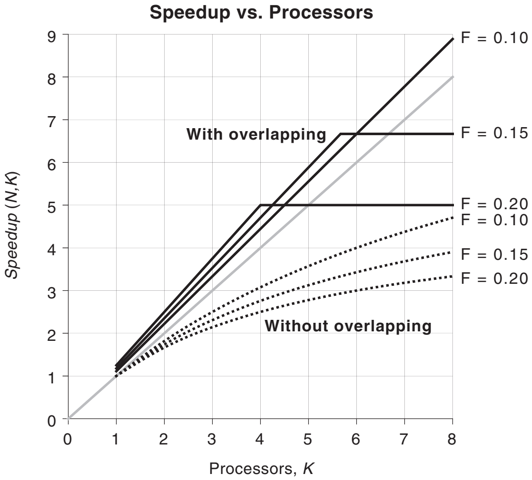


**Figure 18.2** Predicted speedup with overlapping

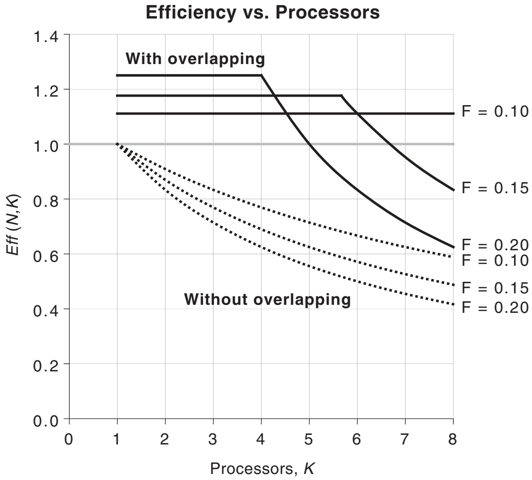


**Figure 18.3** Predicted efficiency with overlapping

Overlapping is a way to get linear or even superlinear speedup behavior for larger numbers of processors than is possible without overlapping. If the sequential fraction is small enough, the program can even exhibit linear speedups all the way up to the maximum number of processors in the SMP parallel computer, as can be seen for $F = 0.10$ in Figure 18.2. However, overlapping cannot cheat Amdahl's Law. Once the speedup hits its limit, adding more processors does not decrease the running time further.

## 18.2 Parallel Sections

Hitherto, all our Parallel Java programs had every parallel team thread executing the *same* piece of code, namely the body of a parallel loop (although with different loop indexes in each thread). To write a Parallel Java program that uses overlapping, we need a parallel team of threads where each thread is executing a *different* piece of code. For the CCA program, we need to execute two sections of code in parallel: the computation section that calculates the cells' next states, and the I/O section that computes and writes the current cells' pixel data to the image file. The relevant Parallel Java construct is the **parallel section**.

To execute multiple sections of code in parallel, first create a parallel team executing a parallel region, as usual. Typically, we want as many team threads as there are sections, so we specify the number of threads explicitly to the parallel team constructor. However, the number of team threads need not be the same as the number of sections. If there are more sections than there are physical processors, for example, we'd create a parallel team with as many threads as there are processors (using class ParallelTeam's no-argument constructor).

```
new ParallelTeam(2).execute (new ParallelRegion()
   {
   . . .
   });
```

Inside the parallel region's `run()` method, call the `execute()` method and pass one or more parallel section objects—instances of class ParallelSection—as the arguments. (In this example, we are doing two parallel sections, but you can specify any number of parallel sections.)

```
new ParallelTeam(2).execute (new ParallelRegion()
   {
   public void run()
      {
      execute (new ParallelSection()
         {
         . . .
         },
      new ParallelSection()
         {
         . . .
         });
      }
   });
```

Put the code for each section in the parallel sections' `run()` methods.

```
new ParallelTeam(2).execute (new ParallelRegion()
   {
   public void run()
      {
      execute (new ParallelSection()
         {
         public void run()
            {
            // Code for computation section goes here
            }
         },

      new ParallelSection()
         {
         public void run()
            {
            // Code for I/O section goes here
            }
         });
      }
   });
```

If needed, add a barrier action argument after the parallel section arguments in the `execute()` method call. As with a parallel for loop, the barrier action can be an instance of class BarrierAction, the constant `BarrierAction.WAIT` (which is the default if the barrier action is omitted), or the constant `BarrierAction.NO_WAIT`.

```
new ParallelTeam(2).execute (new ParallelRegion()
   {
   public void run()
      {
      execute (new ParallelSection()
         {
         public void run()
            {
            // Code for computation section goes here
            }
         },
      new ParallelSection()
         {
         public void run()
            {
            // Code for I/O section goes here
            }
         },
      new BarrierAction()
         {
         public void run()
            {
            // Code for single-threaded barrier action goes here
            }
         });
      }
   });
```

Here's what happens under the hood (Figure 18.4). The parallel team threads all call the parallel region's `run()` method and all in turn call the parallel region's `execute()` method, passing in the parallel section objects. The parallel region now hands off a different section to each thread. Each thread calls its own section's `run()` method. Thus, the sections' `run()` methods are executed in parallel by different threads. If there are more sections than threads, as each thread finishes executing the previous section, the parallel region hands off a new section to the thread. When there are no more sections to execute, each thread waits at a barrier. When all sections have finished executing and all threads have arrived at the barrier, one thread calls the barrier action's `run()` method (if there is a barrier action) while the other threads continue to wait. When the barrier action, if any, finishes executing, all the parallel team threads resume, the parallel region's `execute()` method returns, and each thread continues executing whatever comes next in the parallel region's `run()` method.

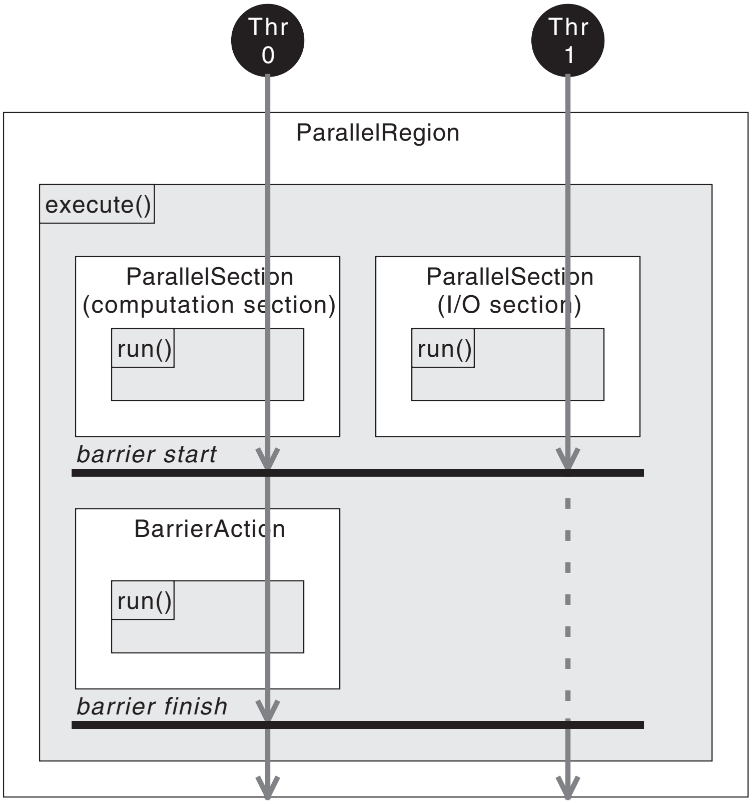


**Figure 18.4** A parallel team executing a group of parallel sections

## 18.3 Nested Parallel Regions

Applying the parallel sections pattern to the design of the CCA program, we compute the cells' next states at the same time as we output the cells' current states. The barrier after the parallel sections prevents the program from going to the next step in the evolution until both the computation section and the I/O section have finished the current step. Swapping the cell array references must still be done in a single-threaded barrier action, though. The new parallel CCA program's design follows Figure 18.4.

```
For $c$ = 0 to $C$–1:
    $currCell_c \leftarrow 0$
$currCell_{C/2} \leftarrow 1$
Parallel region:
    For $s$ = 0 to $S$–1:
        Parallel sections: (two threads)
            Compute all elements of $nextCell$ from $currCell$
            Compute and write $pixel$ data as row $s$ of PJG image file
        Barrier action: (single thread)
            Swap ($currCell$, $nextCell$)
Compute and write $pixel$ data as row $S$ of PJG image file
```

This is an example of the **specialist parallelism** design pattern from Chapter 3. The specialists are the two parallel team threads executing the two parallel sections. One is the computation specialist; the other is the I/O specialist.

The computation specialist's task, however, is itself a result parallel problem, that of computing all the cells' next states. So the computation section should itself contain a parallel region executed by a multiple-thread parallel team. The parallel CCA program's design has **nested parallel regions** (Figure 18.5).

```
For $c = 0$ to $C–1$:
    $currCell_c \leftarrow 0$
$currCell_{C/2} \leftarrow 1$
Parallel region:
    For $s = 0$ to $S–1$:
        Parallel sections: (two threads)
            Parallel region:
                Parallel for $c = 0$ to $C–1$: ($K$ threads)
                    $nextCell_c \leftarrow$ frac(($currCell_{c-1} + currCell_c +$
                        $currCell_{c+1}) \times 1/3 \times A + B$)
            Compute and write $pixel$ data as row $s$ of PJG image file
        Barrier action: (single thread)
            Swap ($currCell$, $nextCell$)
Compute and write $pixel$ data as row $S$ of PJG image file
```

The program has two parallel teams, an outer team with two threads and an inner team with $K$ threads. The outer team executes the computation section and the I/O section in parallel. The computation thread has the inner team execute the parallel for loop. The computation thread waits for the inner team to finish the parallel for loop. Then the computation thread synchronizes with the I/O thread at the barrier, the barrier action is performed, and the program moves to the next step in the CCA evolution.

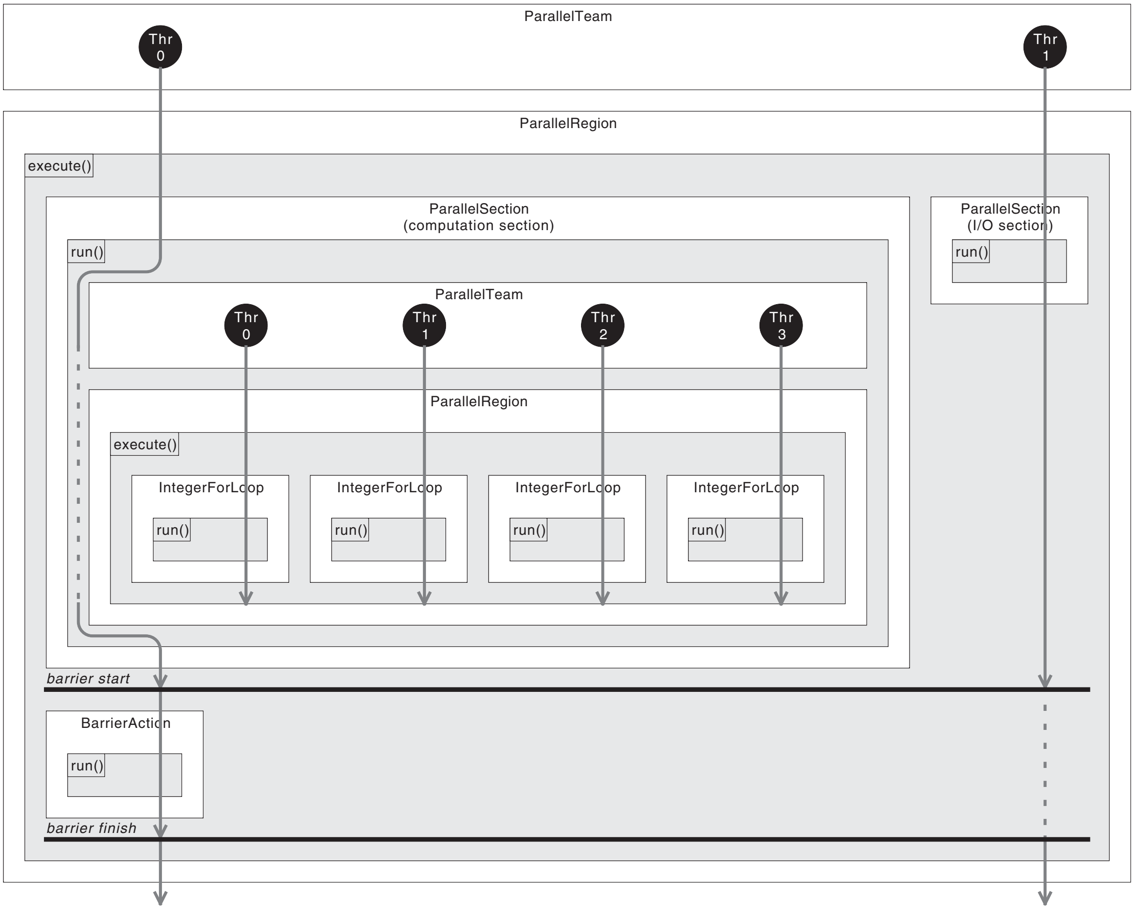


**Figure 18.5** Parallel CCA program design with nested parallel regions

## 18.4 Parallel Program with Overlapping

Here is the source code for class edu.rit.smp.ca.CCASmp2, the parallel CCA program with overlapped computation and I/O, following the design in Figure 18.5.

```
package edu.rit.smp.ca;
import edu.rit.image.GrayImageRow;
import edu.rit.image.PJGGrayImage;
import edu.rit.image.PJGImage;
import edu.rit.pj.BarrierAction;
import edu.rit.pj.IntegerForLoop;
import edu.rit.pj.IntegerSchedule;
import edu.rit.pj.ParallelRegion;
import edu.rit.pj.ParallelSection;
import edu.rit.pj.ParallelTeam;
import edu.rit.util.Range;
import java.io.BufferedOutputStream;
import java.io.File;
import java.io.FileOutputStream;
import java.io.IOException;
public class CCASmp2
   {
   // Constants.
   static final BigRational ZERO = new BigRational ("0");
   static final BigRational ONE = new BigRational ("1");
   static final BigRational ONE_THIRD = new BigRational ("1/3");

   // Command line arguments.
   static int C;
   static int S;
   static BigRational A;
   static BigRational B;
   static File imagefile;

   // Old and new cell arrays.
   static BigRational[] currentCell;
   static BigRational[] nextCell;

   // Grayscale image matrix.
   static byte[][] pixelmatrix;
   static PJGGrayImage image;
   static PJGImage.Writer writer;

   // One row of the grayscale image matrix.
   static byte[] pixelrow;
   static GrayImageRow imagerow;

   // Thread team for cell calculations.
   static ParallelTeam calcTeam;

   /**
    * Main program.
    */
   public static void main
      (String[] args)
      throws Exception
      {
      // Start timing.
      long t1 = System.currentTimeMillis();
```

```
         // Parse command line arguments.
         if (args.length != 5) usage();
         int argi = 0;
         C = Integer.parseInt (args[0]);
         S = Integer.parseInt (args[1]);
         A = new BigRational (args[2]) .mul (ONE_THIRD);
         B = new BigRational (args[3]);
         imagefile = new File (args[4]);

         // Allocate storage for old and new cell arrays. Initialize
         // all cells to 0, except center cell to 1.
         currentCell = new BigRational [C];
         nextCell = new BigRational [C];
         for (int i = 0; i < C; ++ i)
            {
            currentCell[i] = new BigRational();
            nextCell[i] = new BigRational();
            }
         currentCell[C/2].assign (ONE);

         // Set up pixel matrix, image, and image writer.
         pixelmatrix = new byte [S+1] [];
         image = new PJGGrayImage (S+1, C, pixelmatrix);
         writer =
            image.prepareToWrite
               (new BufferedOutputStream
                  (new FileOutputStream (imagefile)));

         // Allocate storage for one pixel matrix row.
         pixelrow = new byte [C];
         imagerow = new GrayImageRow (pixelrow);
         imagerow.setInterpretation (PJGGrayImage.ZERO_IS_WHITE);
```

Here, we create the inner parallel team with *K* threads, *K* being specified with the `-Dpj.nt` flag on the command line. We create this parallel team once and re-use it inside the computation section on each outer loop iteration.

```
         // Set up thread team for cell calculations.
         calcTeam = new ParallelTeam();
```

And here is the outer parallel team with two threads, executing the outer parallel region.

```
         // Perform overlapped computation and I/O.
         new ParallelTeam(2).execute (new ParallelRegion()
            {
            public void run() throws Exception
               {
               // Do S time steps. Sequential outer loop.
               for (int s = 0; s < S; ++ s)
                  {
                  final int step = s;

                  // Calculation section.
                  execute (new ParallelSection()
                     {
                     public void run() throws Exception
                        {
                        calculateNextCell();
                        }
                     },

                  // I/O section.
                  new ParallelSection()
                     {
                     public void run() throws Exception
                        {
                        // I/O section.
                        writeCurrentCell (step);
                        }
                     },

                  // Synchronize threads before next outer loop
                  // iteration.
                  new BarrierAction()
                     {
                     public void run() throws Exception
                        {
                        // Advance one time step — swap old and new
                        // cell arrays.
                        BigRational[] tmp = currentCell;
                        currentCell = nextCell;
                        nextCell = tmp;
                        }
                     });
                  }
               }
            });

         // Write final CA state to image file.
         writeCurrentCell (S);
```

```
   writer.close();

   // Stop timing.
   long t2 = System.currentTimeMillis();
   System.out.println ((t2-t1) + " msec total");
   }
```

The `calculateNextCell()` subroutine contains the code for the calculation section. The inner parallel team executes the inner parallel region, which executes the parallel for loop.

```
/**
 * Calculate the next state of each cell in parallel.
 */
private static void calculateNextCell()
   throws Exception
   {
   // Calculate next state of each cell. Parallel inner loop.
   calcTeam.execute (new ParallelRegion()
      {
      public void run() throws Exception
         {
         execute (0, C-1, new IntegerForLoop()
            {
            public IntegerSchedule schedule()
               {
               return IntegerSchedule.guided();
               }
            public void run (int first, int last)
               {
               for (int i = first; i <= last; ++ i)
                  {
                  nextCell[i]
                     .assign (currentCell[i])
                     .add (currentCell[(i-1+C)%C])
                     .add (currentCell[(i+1)%C])
                     .mul (A)
                     .add (B)
                     .normalize()
                     .fracPart();
                  }
               }
            },
         BarrierAction.NO_WAIT);
         }
         });
      }
```

The inner parallel team threads do not need to wait at a barrier after the parallel for loop, because the barrier after the parallel sections in the outer parallel region suffices. The `writeCurrentCell()` subroutine contains the code for the I/O section.

```
   /**
    * Write the current cell array to the given row of the image
    * file.
    */
   private static void writeCurrentCell
      (int r)
      throws IOException
      {
      // Set image row's gray values based on current cell states.
      for (int i = 0; i < C; ++ i)
         {
         imagerow.setPixel (i, currentCell[i].floatValue());
         }

      // Set row r of the pixel matrix.
      pixelmatrix[r] = pixelrow;

      // Write row-r slice of the image to the image file.
      writer.writeRowSlice (new Range (r, r));
      }
   }
```

Table 18.1 (at the end of the chapter) lists, and Figure 18.6 plots, the CCASmp2 program's performance on the "parasite" SMP parallel computer. The command-line arguments are the same as in Chapter 17: $C$ = 2,000 cells; $S$ = 400, 500, 600, 700, 800, and 1,000 steps; $A$ = 1; and $B$ = 11/12.

There's one slight anomaly in the data. The running time increases, so the speedup decreases, when going from $K$ = 7 to $K$ = 8. This is because $K$ is the number of *computation* threads; the I/O is being done in an additional thread. However, the "parasite" computer has only eight physical processors. Thus, when $K$ is 7, the program is using all eight processors. When $K$ is 8, the JVM is trying to run nine threads on eight processors, and two of the threads must share the same processor. Because the operating system swaps back and forth between these two threads, they do not have full use of the processor, they take longer to finish, and the program's overall running time increases.

The measured speedup and efficiency curves in Figure 18.6 don't look like the theoretical speedup and efficiency curves in Figures 18.2–18.3. This is because the CCASmp2 program only overlaps the parallel portion with *some* of the sequential portion, namely computing the pixels' gray values and writing them to the image file. The rest of the sequential portion, namely swapping the cell array references, is still performed in a single thread. The necessary barrier synchronization among the threads also is not overlapped with the parallel portion. Figures 18.2–18.3 depict the ideal situation where *all* of the sequential portion overlaps the parallel portion.

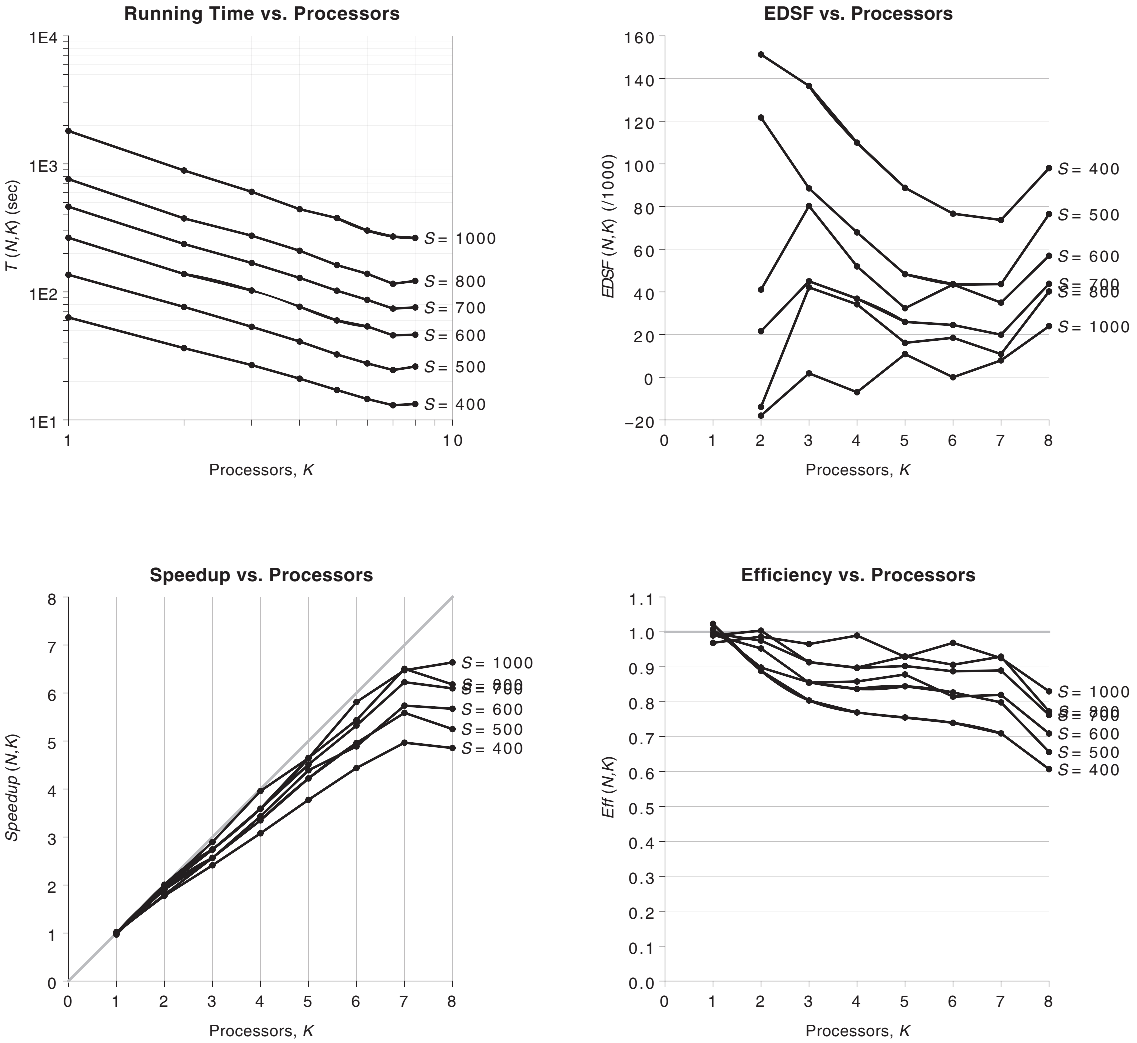


**Figure 18.6** CCASeq/CCASmp2 running-time metrics

So has the overlapping done any good? Figure 18.7 plots the running-time metrics for $S = 800$ for both the CCASmp program (without overlapping) and the CCASmp2 program (with overlapping) on the same axes. Overlapping has clearly reduced the running times, reduced the *EDSFs,* increased the speedups, and increased the efficiencies (except when $K = 8$). Similar improvements in the metrics can be seen for the other problem sizes as well. While the improvements on this program were modest, other programs—those with larger sequential fractions due to I/O—might see more dramatic improvements using overlapping.

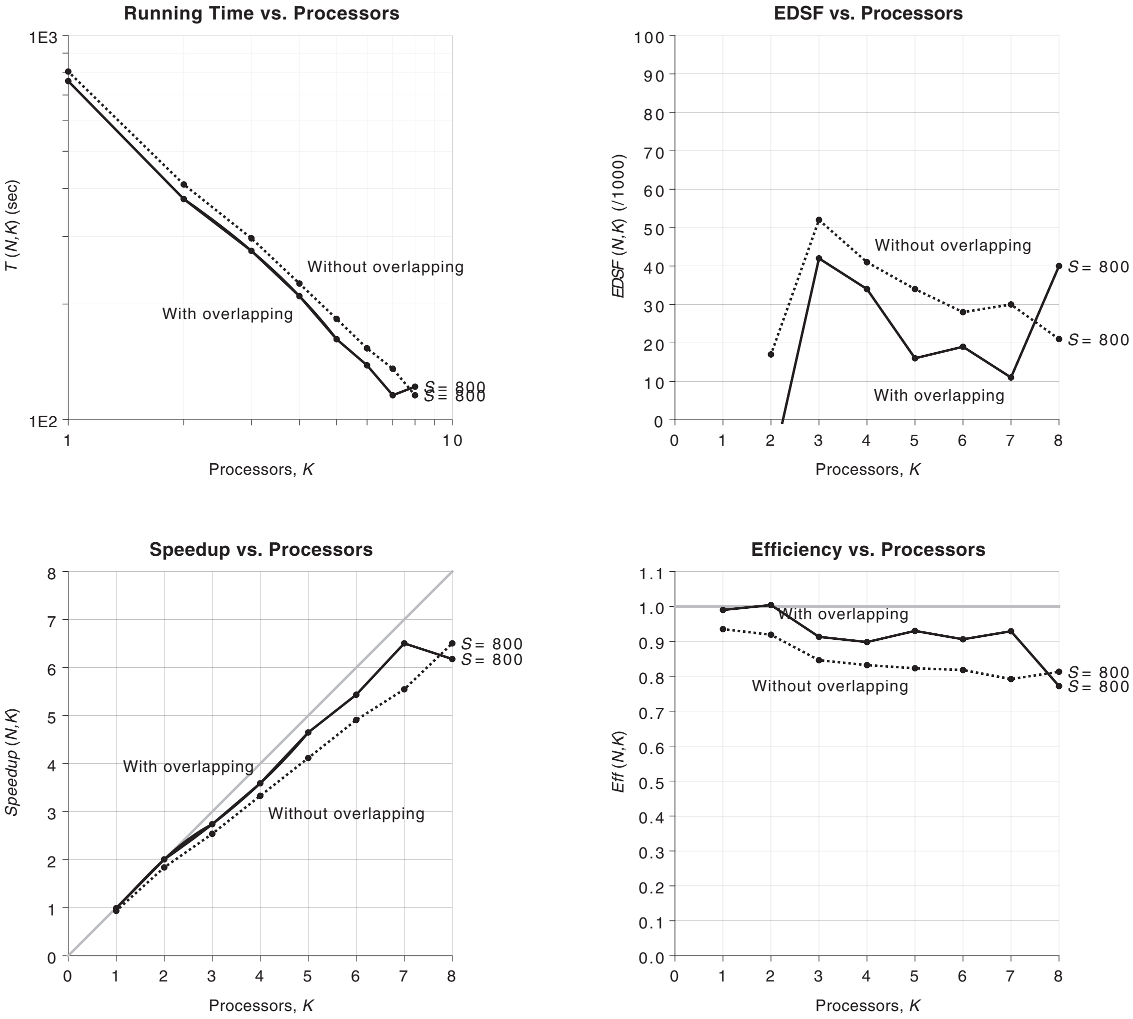


**Figure 18.7** Running-time metrics with and without overlapping

**Table 18.1** CCASeq/CCASmp2 running-time metrics

| *S* | *K* | *T* | *Spdup* | *Eff* | *EDSF* | *S* | *K* | *T* | *Spdup* | *Eff* | *EDSF* |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 400 | seq | 64678 | | | | 700 | seq | 461504 | | | |
| 400 | 1 | 63215 | 1.023 | 1.023 | | 700 | 1 | 463389 | 0.996 | 0.996 | |
| 400 | 2 | 36389 | 1.777 | 0.889 | 0.151 | 700 | 2 | 236698 | 1.950 | 0.975 | 0.022 |
| 400 | 3 | 26826 | 2.411 | 0.804 | 0.137 | 700 | 3 | 168363 | 2.741 | 0.914 | 0.045 |
| 400 | 4 | 21018 | 3.077 | 0.769 | 0.110 | 700 | 4 | 128651 | 3.587 | 0.897 | 0.037 |
| 400 | 5 | 17134 | 3.775 | 0.755 | 0.089 | 700 | 5 | 102300 | 4.511 | 0.902 | 0.026 |
| 400 | 6 | 14575 | 4.438 | 0.740 | 0.077 | 700 | 6 | 86696 | 5.323 | 0.887 | 0.025 |
| 400 | 7 | 13025 | 4.966 | 0.709 | 0.074 | 700 | 7 | 74125 | 6.226 | 0.889 | 0.020 |
| 400 | 8 | 13323 | 4.855 | 0.607 | 0.098 | 700 | 8 | 75694 | 6.097 | 0.762 | 0.044 |
| 500 | seq | 137161 | | | | 800 | seq | 753220 | | | |
| 500 | 1 | 136173 | 1.007 | 1.007 | | 800 | 1 | 760990 | 0.990 | 0.990 | |
| 500 | 2 | 76374 | 1.796 | 0.898 | 0.122 | 800 | 2 | 375233 | 2.007 | 1.004 | -0.014 |
| 500 | 3 | 53429 | 2.567 | 0.856 | 0.089 | 800 | 3 | 275040 | 2.739 | 0.913 | 0.042 |
| 500 | 4 | 40978 | 3.347 | 0.837 | 0.068 | 800 | 4 | 209763 | 3.591 | 0.898 | 0.034 |
| 500 | 5 | 32497 | 4.221 | 0.844 | 0.048 | 800 | 5 | 162022 | 4.649 | 0.930 | 0.016 |
| 500 | 6 | 27655 | 4.960 | 0.827 | 0.044 | 800 | 6 | 138575 | 5.435 | 0.906 | 0.019 |
| 500 | 7 | 24550 | 5.587 | 0.798 | 0.044 | 800 | 7 | 115803 | 6.504 | 0.929 | 0.011 |
| 500 | 8 | 26133 | 5.249 | 0.656 | 0.076 | 800 | 8 | 121917 | 6.178 | 0.772 | 0.040 |
| 600 | seq | 262908 | | | | 1000 | seq | 1753323 | | | |
| 600 | 1 | 265109 | 0.992 | 0.992 | | 1000 | 1 | 1809869 | 0.969 | 0.969 | |
| 600 | 2 | 138001 | 1.905 | 0.953 | 0.041 | 1000 | 2 | 888667 | 1.973 | 0.986 | -0.018 |
| 600 | 3 | 102564 | 2.563 | 0.854 | 0.080 | 1000 | 3 | 605515 | 2.896 | 0.965 | 0.002 |
| 600 | 4 | 76610 | 3.432 | 0.858 | 0.052 | 1000 | 4 | 442961 | 3.958 | 0.990 | -0.007 |
| 600 | 5 | 59885 | 4.390 | 0.878 | 0.032 | 1000 | 5 | 377692 | 4.642 | 0.928 | 0.011 |
| 600 | 6 | 53791 | 4.888 | 0.815 | 0.043 | 1000 | 6 | 301661 | 5.812 | 0.969 | 0.000 |
| 600 | 7 | 45821 | 5.738 | 0.820 | 0.035 | 1000 | 7 | 270836 | 6.474 | 0.925 | 0.008 |
| 600 | 8 | 46353 | 5.672 | 0.709 | 0.057 | 1000 | 8 | 264149 | 6.638 | 0.830 | 0.024 |

# PART II

# Exercises

1. A parallel program has a parallel loop with *N* iterations and runs on an SMP parallel computer with *K* processors (threads). Write a short program whose inputs are *N* and *K* and whose outputs are the lower and upper bounds of each thread's range of loop indexes, such that the total set of loop indexes (0 through *N*–1) is partitioned as equally as possible among the *K* threads. Keep in mind that each index bound must be an integer and that *N* need not be evenly divisible by *K*.
2. In the AES partial key search program searching for *n* missing key bits and running on *K* parallel processors, give the lower and upper bounds of each thread's range of loop indexes for $n = 16$, 17, 18, and 19 and for $K = 2$, 3, 4, and 5.
3. Suppose a computer takes 2.5 microseconds to perform one AES encryption. What will the FindKeySeq program's running time be for $n = 16$? $n = 20$? $n = 24$? $n = 28$? Neglect all times other than the AES encryption time.
4. Why is the AES partial key search problem classified as an agenda parallel problem rather than a result parallel problem?
5. How long does the FindKeySeq program take on your computer for various values of *n*?
6. Approximately how long does your computer take to perform one AES encryption using class AES256Cipher?

7. How do the FindKeySeq program’s running times compare with the FindKeySmp program’s running times on your computer with identical inputs when using one parallel team thread (`-Dpj.nt=1`)? If there is any difference, what causes it?

8. Write an SMP parallel program fragment using Parallel Java to compute the vector sum *S* of two vectors *A* and *B*. *A*, *B*, and *S* are *n*-element vectors of double-precision floating-point numbers. Each element of *S* is the sum of the corresponding elements of *A* and *B*. What parallel design pattern or patterns does this problem exhibit?

9. Write an SMP parallel program fragment using Parallel Java to compute the dot product *C* of two vectors *A* and *B*. *A* and *B* are *n*-element vectors of double-precision floating-point numbers. *C* is a double-precision floating-point number given by the following formula:

$$C = \sum_{i=0}^{n-1} A_i \cdot B_i \tag{1}$$

   What parallel design pattern or patterns does this problem exhibit?

*Exercises 10–14.* The following is a fragment of a sequential Java program to compute a table of the sine function, sin(*x*), for the values *x* = 0.0, 0.1, 0.2, . . ., 15.6, 15.7, using the Taylor series expansion:

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \ldots \tag{2}$$

Each value of the sine function is to be computed with three digits of precision. Note that for different values of *x*, different numbers of terms in the Taylor series might be needed to achieve the required precision.

```
double[] sineTable = new double [158];
sineTable[0] = 0.0;
for (int i = 1; i <= 157; ++ i)
   {
   double x = i / 10.0;
   double sum = 0.0;
   double term = x;
   int denom = 1;
   do
      {
      sum += term;
      term = -term * x * x / (denom+1) / (denom+2);
      denom += 2;
      }
   while (Math.abs (term / sum) >= 0.001);
```

```
      sineTable[i] = sum;
      }
```

10. Describe the sequential dependencies, if any, in the preceding program. Is it possible to parallelize the outer for loop? Is it possible to parallelize the inner do-while loop?
11. Write an SMP parallel program fragment using Parallel Java to compute the sine function table.
12. Which parallel design pattern or patterns does your program use?
13. In the preceding program fragment, explain why the line "`sineTable[0] = 0.0;`" is there.
14. The parallel program is required to have a balanced load. To achieve a balanced load, do you have to add any statements to the program? If not, explain why not. If so, describe the statements that must be added.

*Exercises 15–18.* The following is a fragment of a sequential Java program to compute the outer product *C* of two vectors *A* and *B*. *A* and *B* are *n*-element vectors, and *C* is an *n*×*n* matrix defined as follows:

$$C_{ij} = A_i \cdot B_j,\ i, j = 0 \ldots N-1 \qquad (3)$$

```
   int n = ...;
   double[] a = new double [n];
   double[] b = new double [n];
   double[][] c = new double [n] [n];
   for (int i = 0; i < n; ++ i)
      {
      for (int j = 0; j < n; ++ j)
         {
         c[i][j] = a[i] * b[j];
         }
      }
```

15. Describe the sequential dependencies, if any, in the preceding program. Is it possible to parallelize the outer for loop? Is it possible to parallelize the inner for loop?
16. Write an SMP parallel program fragment using Parallel Java to compute the outer product.
17. Which parallel design pattern or patterns does your program use?

18. The parallel program is required to have a balanced load. To achieve a balanced load, do you have to add any statements to the program? If not, explain why not. If so, describe the statements that must be added.

19. A certain program initializes the variable $x$ to a fixed value, and then reads a sequence of integers from an input file. For each integer $i$ in the input file, if $i = 1$, the program sets $x = f_1(x)$; if $i = 2$, the program sets $x = f_2(x)$; if $i = 3$, the program sets $x = f_3(x)$; if $i = 4$, the program sets $x = f_4(x)$; if $i$ is anything else, the program prints an error message and exits. After the program has processed the entire input file, the program prints the final value of $x$ and exits. Because the input file can be quite large and each function $f_1$, $f_2$, $f_3$, and $f_4$ can take some time to compute, we want to implement this program as a parallel program and run it on multiple processors to get a speedup. Is it possible for a parallel version of this program to yield a speedup, compared to a sequential version of this program? Explain why or why not.

20. The **binomial coefficient** $B(n,k)$ is defined by the following formula:

$$B(n,k) = \frac{n!}{k!\ (n-k)!} \tag{4}$$

If you make a table of the binomial coefficient for various values of $n$ and $k$, you get **Pascal's Triangle**:

| | *k*= | | | | | |
|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 | … |
| *n*=0 | 1 | 0 | 0 | 0 | 0 | … |
| 1 | 1 | 1 | 0 | 0 | 0 | … |
| 2 | 1 | 2 | 1 | 0 | 0 | … |
| 3 | 1 | 3 | 3 | 1 | 0 | … |
| 4 | 1 | 4 | 6 | 4 | 1 | … |
| ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋱ |

Suppose you want to compute a very large table of Pascal's Triangle. Computing each table entry using the preceding formula for $B(n,k)$ would require many multiplications and divisions. However, note that for $n > 0$ and $k > 0$, $B(n,k) = B(n-1,k-1) + B(n-1,k)$. This suggests a sequential algorithm for computing each table entry simply by adding two entries in the previous row. Is it possible to modify that sequential algorithm to get a parallel algorithm for an SMP parallel computer? If so, describe the parallel algorithm. If not, explain why a parallel algorithm is impossible.

*Exercises 21–25.* The year is 1944. World War II is at its height. The Computer Department at your company has 50 state-of-the-art computers all together in a big room. Each "computer" is a young lady with a Friden automatic electromechanical calculator that can do addition, subtraction, multiplication, and division. A computer takes the following amount of time to perform a calculation: 1 second to add two numbers or subtract two numbers, 5 seconds to multiply two numbers or divide two numbers. Your task is to prepare a table of the function

$$f(x) = ax^2 + bx + c \tag{5}$$

for given constant values $a$, $b$, and $c$ and for 1,020 values of $x$: $x = 0.1, 0.2, 0.3, \ldots, 101.8, 101.9, 102.0$. This table will be used for computing the trajectories of artillery shells on the battlefield.

21. The mathematical expert at your company has recommended that you calculate the function using Equation (6), which is equivalent to Equation (5). Explain why the expert recommends using the alternate formula.

$$f(x) = (ax + b)x + c \tag{6}$$

22. Considering only calculation time, how long will it take for one computer to do the task using Equation (6)?

$P \leq \sqrt{N} < 1000$
$P \leq \sqrt{N} < 1000$
$P \leq \sqrt{N} < 1000$

23. Considering only calculation time, how long will it take for the whole Computer Department to do the task in parallel using Equation (6)?
24. Considering only calculation time, what is the speedup of the whole Computer Department for this task?
25. Considering only calculation time, what is the efficiency of the whole Computer Department for this task?

*Exercises 26–29.* Here's another problem for the Computer Department. Each computer has a sheet of paper with a list of all the prime numbers less than or equal to 1000; there are 168 primes in the list. Your problem is to determine whether the number $N = 988027$ is prime. The algorithm is as follows: Divide $N$ by each prime in the list; if any prime in the list divides $N$ with no remainder, then $N$ is not prime; otherwise $N$ is prime. The algorithm always goes through the entire list of primes. (Note that if $N$ is not prime, then $N$ must have a prime factor $P \leq \sqrt{N} < 1000$, so $P$ must be in the list of primes and the algorithm is guaranteed to find $P$.)

26. Considering only calculation time, how long will it take for one computer to solve the problem?
27. Considering only calculation time, how long will it take for the whole Computer Department to solve the problem in parallel?
28. Considering only calculation time, what is the speedup of the whole Computer Department for this problem?
29. Considering only calculation time, what is the efficiency of the whole Computer Department for this problem?

*Exercises 30–34.* Here's another problem for the Computer Department. Multiply two matrices $A$ and $B$ to get the product matrix $C = A{\cdot}B$. Each matrix has 12 rows and 12 columns of numbers. The elements of the two matrices $A$ and $B$ have been printed on a sheet of paper, and each computer has a copy. The formula for computing the element at row $i$, column $j$ of the product is the following:

$$C_{ij} = \sum_{k=1}^{12} A_{ik} \cdot B_{kj} \tag{7}$$

30. Considering only calculation time, how long will it take for one computer to solve the problem?
31. Considering only calculation time, how long will it take for the whole Computer Department to solve the problem in parallel?
32. Considering only calculation time, what is the speedup of the whole Computer Department for this problem?
33. Considering only calculation time, what is the efficiency of the whole Computer Department for this problem?
34. Your boss proposes to buy 10 more Friden calculators and to hire and train 10 more young ladies to solve the problem faster. Will this expansion of the Computer Department in fact solve the problem faster? Explain why or why not.

*Exercises 35–38.* Here's another problem for the Computer Department. Compute the mean of 256 given numbers $x_1$ through $x_{256}$; that is:

$$\frac{1}{256}\sum_{i=1}^{256} x_i \tag{8}$$

35. Considering only calculation time, how long will it take for one computer to solve the problem?
36. Considering only calculation time, how long will it take for the whole Computer Department to solve the problem in parallel?
37. Considering only calculation time, what is the speedup of the whole Computer Department for this problem?
38. Considering only calculation time, what is the efficiency of the whole Computer Department for this problem?

*Exercises 39–42.* Here's another problem for the Computer Department. Your company's Sales Department has 360 salesmen (in 1944, they were all males) selling floor brushes, hairbrushes, toothbrushes, and scrub brushes door-to-door. Each salesman has submitted his quarterly sales report showing the number of floor brushes, hairbrushes, toothbrushes, and scrub brushes he sold. Your task is to calculate each salesman's total commission. The commission is 50 cents for each floor brush, 25 cents for each hairbrush, 10 cents for each toothbrush, and one dollar for each scrub brush.

39. Considering only calculation time, how long will it take for one computer to solve the problem?
40. Considering only calculation time, how long will it take for the whole Computer Department to solve the problem in parallel?
41. Considering only calculation time, what is the speedup of the whole Computer Department for this problem?
42. Considering only calculation time, what is the efficiency of the whole Computer Department for this problem?

*Exercises 43–47.* You have made the following measurements of running time $T$ (msec) versus number of processors $K$ for a certain parallel program solving a problem with a fixed problem size $N$:

| $K$ | $T$ |
|---|---|
| 1 | 14000 |
| 2 | 7560 |
| 3 | 5413 |
| 4 | 4340 |

43. What fraction of this program's total running time must be performed sequentially?
44. What would the program's running time be for $K = 5$ processors?
45. What are the program's speedup and efficiency for $K = 1, 2, 3, 4$, and 5 processors?
46. Suppose all of the program's sequential portion is overlapped with the parallel portion. What are the program's speedup and efficiency for $K = 1, 2, 3, 4$, and 5 processors?
47. What is the maximum speedup you can expect from this program as $K$ increases while holding $N$ fixed?

*Exercises 48–52.* You have made the following measurements of running time $T$ (msec) versus number of processors $K$ for a certain parallel program solving a problem with a fixed problem size $N$:

| $K$ | $T$ |
|---|---|
| 1 | 50000 |
| 2 | 28750 |
| 3 | 21667 |
| 4 | 18125 |

48. What fraction of this program's total running time must be performed sequentially?
49. What would the program's running time be for $K = 5$ processors?

50. What are the program's speedup and efficiency for $K = 1, 2, 3, 4$, and 5 processors?

51. Suppose all of the program's sequential portion is overlapped with the parallel portion. What are the program's speedup and efficiency for $K = 1, 2, 3, 4$, and 5 processors?

52. What is the maximum speedup you can expect from this program as $K$ increases while holding $N$ fixed?

*Exercises 53–57.* You have made the following measurements of running time $T$ (msec) versus number of processors $K$ for a certain parallel program solving a problem with a fixed problem size $N$:

| $K$ | $T$ |
|---|---|
| 1 | 240000 |
| 2 | 134400 |
| 3 | 99200 |
| 4 | 81600 |

53. What fraction of this program's total running time must be performed sequentially?

54. What would the program's running time be for $K = 8$ processors?

55. What are the program's speedup and efficiency for $K = 1, 2, 3, 4$, and 8 processors?

56. Suppose all of the program's sequential portion is overlapped with the parallel portion. What are the program's speedup and efficiency for $K = 1, 2, 3, 4$, and 8 processors?

57. What is the maximum speedup you can expect from this program as $K$ increases while holding $N$ fixed?

*Exercises 58–62.* You have measured the running time $T$ for a certain program for several values of $n$, where the problem size $N = n^3$, and for several values of $K$, the number of processors:

| $n$ | $K$ | $T$ | $n$ | $K$ | $T$ | $n$ | $K$ | $T$ | $n$ | $K$ | $T$ |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 100 | seq | 10926 | 125 | seq | 20966 | 150 | seq | 35964 | 200 | seq | 83368 |
| 100 | 1 | 11315 | 125 | 1 | 21530 | 150 | 1 | 36366 | 200 | 1 | 81626 |
| 100 | 2 | 6194 | 125 | 2 | 10841 | 150 | 2 | 18764 | 200 | 2 | 42750 |
| 100 | 3 | 4376 | 125 | 3 | 7669 | 150 | 3 | 12383 | 200 | 3 | 27822 |
| 100 | 4 | 3543 | 125 | 4 | 6155 | 150 | 4 | 9806 | 200 | 4 | 21164 |

58. Calculate *Speedup*, *Eff*, and *EDSF* as a function of $K$ for each value of $N$.

59. Calculate $N$, *Sizeup*, and *SizeupEff* as a function of $K$ for $T = 15{,}000$.

60. Calculate $N$, *Sizeup*, and *SizeupEff* as a function of $K$ for $T = 20{,}000$.

61. Calculate the Second Problem Size Law model parameters for this program. Use the TimeFit program in the Parallel Java Library.

62. What is the maximum sizeup this program can achieve as $K$ increases?

*Exercises 63–67.* You have measured the running time $T$ for a certain program for several values of $N$, the problem size, and for several values of $K$, the number of processors:

| $N$ | $K$ | $T$ | $N$ | $K$ | $T$ | $N$ | $K$ | $T$ | $N$ | $K$ | $T$ | $N$ | $K$ | $T$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 40 | seq | 2377 | 160 | seq | 7812 | 360 | seq | 16855 | 640 | seq | 29534 | 1000 | seq | 45737 |
| 40 | 1 | 2466 | 160 | 1 | 8049 | 360 | 1 | 17165 | 640 | 1 | 30473 | 1000 | 1 | 46864 |
| 40 | 2 | 1516 | 160 | 2 | 4270 | 360 | 2 | 9011 | 640 | 2 | 15474 | 1000 | 2 | 23826 |
| 40 | 3 | 1215 | 160 | 3 | 3081 | 360 | 3 | 6280 | 640 | 3 | 10819 | 1000 | 3 | 16496 |
| 40 | 4 | 1052 | 160 | 4 | 2473 | 360 | 4 | 4896 | 640 | 4 | 8275 | 1000 | 4 | 12634 |

63. Calculate *Speedup*, *Eff*, and *EDSF* as a function of $K$ for each value of $N$.

64. Calculate $N$, *Sizeup*, and *SizeupEff* as a function of $K$ for $T = 5{,}000$.

65. Calculate $N$, *Sizeup*, and *SizeupEff* as a function of $K$ for $T = 10{,}000$.

66. Calculate the Second Problem Size Law model parameters for this program. Use the TimeFit program in the Parallel Java Library.

67. What is the maximum sizeup this program can achieve as $K$ increases?

*Exercises 68–70.* You have made the following measurements of running time $T$ versus number of processors $K$ and problem size $N$ for a certain parallel program. For each value of $N$, you have measured the running time on one processor and on $K$ processors. You have discovered that the running time is constant, provided $N$ scales up with $K$. Assume the First Problem Size Law holds.

| | | One Processor | $K$ Processors |
|---|---|---|---|
| $K$ | $N$ | $T$ | $T$ |
| 1 | 100 | 17250 | 17250 |
| 2 | 200 | 33465 | 17250 |
| 3 | 300 | 49680 | 17250 |
| 4 | 400 | 65895 | 17250 |

68. What fraction of this program's total running time must be performed sequentially for $K = 1$ and $N = 100$?

69. For $K = 5$ and $N = 500$, what will the speedup be?

70. Suppose you don't let $N$ scale up with $K$, but instead you run this program with $K = 2$ and $N = 100$. What will the speedup be?

*Exercises 71–73.* You have made the following measurements of running time $T$ versus number of processors $K$ and problem size $N$ for a certain parallel program. For each value of $N$, you have measured the running time on one processor and on $K$ processors. You have discovered that the running time is constant provided $N$ scales up with $K$. Assume the First Problem Size Law holds.

| | | One Processor | $K$ Processors |
|---|---|---|---|
| $K$ | $N$ | $T$ | $T$ |
| 1 | 100 | 20655 | 20655 |
| 2 | 200 | 40095 | 20655 |
| 3 | 300 | 59535 | 20655 |
| 4 | 400 | 78975 | 20655 |

71. What fraction of this program's total running time must be performed sequentially for $K = 1$ and $N = 100$?

72. For $K = 5$ and $N = 500$, what will the speedup be?

73. Suppose you don't let $N$ scale up with $K$, but instead you run this program with $K = 2$ and $N = 100$. What will the speedup be?

74. Try running the MandelbrotSetSmp2 program with various loop schedules. Examine the data on the starting and ending times of each chunk of loop iterations (see Chapter 12, Section 12.1 and Figure 12.8). What do you learn about how different loop schedules achieve (or fail to achieve) a balanced load?

75. Measure the MandelbrotSetSmp program's running time $T$ using a dynamic schedule for different problem sizes $N$, different numbers of processors $K$, and different chunk sizes. Determine the effect of $N$, $K$, and chunk size on $T$. What dynamic schedule chunk size gives the best performance?

76. Measure the MandelbrotSetSmp program's running time $T$ using a guided schedule for different problem sizes $N$, different numbers of processors $K$, and different minimum chunk sizes. Determine the effect of $N$, $K$, and minimum chunk size on $T$. What guided schedule minimum chunk size gives the best performance? How does this compare with a dynamic schedule?

*Exercises 77–80.* Given an integer $i > 0$, consider the following procedure:

```
$x \leftarrow i$
While $x > 1$:
    If $x$ is even:
        $x \leftarrow x/2$
    Else:
        $x \leftarrow 3x+1$
```

The **Collatz Conjecture**, proposed by Lothar Collatz in 1937, states that for every $i > 0$, the preceding procedure terminates; that is, $x$ eventually becomes 1. While mathematicians believe the Collatz Conjecture is true, no one has been able to prove it.

77. Write a sequential program to investigate whether the Collatz Conjecture is true for all values of $i$ from 1 through $N$, where $N$ is a command-line

argument. Use type `long` so $N$ can be as large as $2^{63}–1$. The program also has a command-line argument *MaxIter* (type `long`). In the preceding procedure, for a certain value of $i$, if $x$ reaches 1 before the number of while loop iterations reaches *MaxIter*, then the Collatz Conjecture is true for $i$. If the number of while loop iterations reaches *MaxIter* before $x$ reaches 1, then the Collatz Conjecture *might* be false for $i$. (The Collatz Conjecture is not *definitely* false for $i$ because $x$ might reach 1 with further iterations, but the program has to stop somewhere.) The program prints the values of $i$ for which the Collatz Conjecture might be false.

78. Describe the sequential dependencies, if any, in the program. Is it possible to parallelize the program?
79. If possible, write an SMP parallel program to investigate the Collatz Conjecture. The parallel program has the same command-line arguments and the same output as the sequential program. Measure the parallel program's running times as a function of $N$ and $K$, calculate the program's running-time metrics, and improve the program's design, if necessary.
80. Do you have to do anything to achieve load balance in the parallel program? If so, describe how to balance the load. If not, explain why not.

*Exercises 81–85.* A **three-dimensional random walk** is defined as follows. A particle is initially positioned at (0, 0, 0) in the X-Y-Z coordinate space. The particle does a sequence of $N$ steps. At each step, the particle chooses one of the six directions left, right, ahead, back, up, or down at random, and then moves one unit in that direction. Specifically, if the particle is at $(x, y, z)$:

With probability 1/6 the particle moves left to $(x–1, y, z)$.
With probability 1/6 the particle moves right to $(x+1, y, z)$.
With probability 1/6 the particle moves back to $(x, y–1, z)$.
With probability 1/6 the particle moves ahead to $(x, y+1, z)$.
With probability 1/6 the particle moves down to $(x, y, z–1)$.
With probability 1/6 the particle moves up to $(x, y, z+1)$.

81. Write a sequential program to calculate the particle's final position. The program's command-line arguments are the random seed and the number of steps $N$. The program prints the particle's final position $(x, y, z)$ as well as the particle's final distance from the origin.
82. Describe the sequential dependencies, if any, in the program. Is it possible to parallelize the program?
83. If possible, write an SMP parallel program to calculate the particle's final position. The parallel program has the same command-line arguments and the same output as the sequential program. Measure the parallel program's running times as a function of $N$ and $K$, calculate the program's running-time metrics, and improve the program's design, if necessary.

84. What is the particle's expected final distance from the origin as a function of the number of steps $N$?

85. Run your program for a large number of steps and a variety of different random seeds. Do the particle's computed final distances from the origin agree with the expected final distance?

86. Run the sequential Monte Carlo π program (class PiSeq) for different numbers of iterations $N$. Compute the fractional error between the program's estimate for π and the true value of π. The theory behind Monte Carlo integration says that the fractional error should be proportional to $\sqrt{N}$ . Is it?

87. Do some research and discover how to modify Floyd's Algorithm to compute the shortest path itself (the sequence of vertices), in addition to the length of the shortest path, between each pair of vertices. Write a sequential program and an SMP parallel program for the modified Floyd's Algorithm and measure their performance. How does their performance compare to the original programs' performance?

88. Like the program for Floyd's Algorithm (FloydSeq/FloydSmpRow), the program that generates an image of the Mandelbrot Set (MandelbrotSetSeq/ MandelbrotSetSmp) computes each element of a large matrix. But the Mandelbrot Set program does not experience an abrupt increase in efficiency as the number of processors increases, as the Floyd's Algorithm program does. Why?

# PART III

# Clusters

*This page intentionally left blank*

CHAPTER 19

# A First Cluster Parallel Program

in which we revisit our introductory sequential program; we convert it to a program for a cluster parallel computer; we see how long it takes to run each version; and we get some insight into how parallel programs execute on a cluster

# CHAPTER 19 A First Cluster Parallel Program

## 19.1 Sequential Program

It’s time to switch from building SMP parallel programs, like those of Figure 2.7, to building cluster parallel programs, like those of Figure 2.9. Before diving in, let’s revisit the simple little program from Chapter 4 that checks its command-line arguments for primality using trial division. Here again is the source code for class Program1Seq, the sequential version.

```
public class Program1Seq
   {
   static int n;
   static long[] x;
   static long t1, t2[], t3[];

   public static void main
      (String[] args)
      throws Exception
      {
      t1 = System.currentTimeMillis();
      n = args.length;
      x = new long [n];
      for (int i = 0; i < n; ++ i)
         {
         x[i] = Long.parseLong (args[i]);
         }
      t2 = new long [n];
      t3 = new long [n];
      for (int i = 0; i < n; ++ i)
         {
         t2[i] = System.currentTimeMillis();
         isPrime (x[i]);
         t3[i] = System.currentTimeMillis();
         }
      for (int i = 0; i < n; ++ i)
         {
         System.out.println
            ("i = "+i+" call start = "+(t2[i]-t1)+" msec");
         System.out.println
            ("i = "+i+" call finish = "+(t3[i]-t1)+" msec");
         }
      }

   private static boolean isPrime
      (long x)
      {
      if (x % 2 == 0) return false;
      long p = 3;
      long psqr = p*p;
      while (psqr <= x)
         {
         if (x % p == 0) return false;
         p += 2;
         psqr = p*p;
         }
      return true;
      }
   }
```

## 19.2 Running the Sequential Program

Here is what Program1Seq printed when run on one backend processor of a cluster parallel computer. In this cluster, each backend processor has a 440 MHz Sun Microsystems UltraSPARC-IIi CPU and 256 MB of main memory.

```
$ java -server Program1Seq 1000000000000037 1000000000000091 \
  1000000000000159 1000000000000187
i = 0 call start = 1 msec
i = 0 call finish = 2662 msec
i = 1 call start = 2662 msec
i = 1 call finish = 5316 msec
i = 2 call start = 5316 msec
i = 2 call finish = 7966 msec
i = 3 call start = 7966 msec
i = 3 call finish = 10611 msec
```

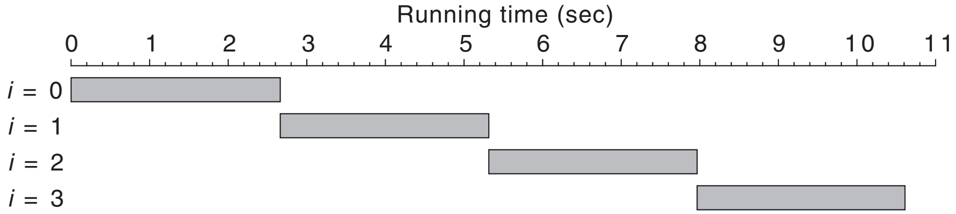


**Figure 19.1** Program1Seq execution timeline, sequential computer

On this computer, the `java` command runs the Java HotSpot Client Virtual Machine by default. The `-server` flag runs the Java HotSpot Server Virtual Machine instead. The Server JVM's JIT compiler does more extensive optimizations than the Client JVM's JIT compiler, thus substantially reducing the program's running time. In general, you should use the Server JVM to run any Java program involving lengthy computation. (On the SMP parallel computers used in the previous chapters, the Server JVM runs by default.)

Plotting each subroutine call's start and finish on a timeline (Figure 19.1) reveals the typical pattern of sequential execution. The program executes each `isPrime()` subroutine call in its entirety before going to the next subroutine call, and there is no parallel execution.

## 19.3 Cluster Parallel Program

Now let's rewrite the program using Parallel Java so it will run in parallel when executed on a cluster parallel computer. Because a cluster parallel program runs on multiple backend processors and sends messages between the backend processors as well as the frontend processor, every cluster parallel program's first act is to initialize Parallel Java's **communication layer**. To do so, we call the `Comm.init()` method, passing in the array of command-line arguments.

```
public static void main
   (String[] args)
   throws Exception
   {
   Comm.init (args);
   }
```

When the `Comm.init()` method returns, the communication layer has created a process on each backend processor of the cluster. Each process is running a JVM, is executing the same `main()` method with the same command-line arguments, and has returned from the `Comm.init()` method call. Each process's communication layer has also created an object called the **world communicator**. A communicator is an object used to send and receive messages, and the world communicator is able to send and receive messages between any or all of the backend processes in the program. Next, we get a reference to the world communicator object, `world`.

```
   static Comm world;

   public static void main
      (String[] args)
      throws Exception
      {
      Comm.init (args);
      world = Comm.world();
      }
```

The world communicator has two attributes, size and rank. The world communicator's **size** is the number of backend processes in the program, *K*. Each process is assigned a different **rank** in the range 0 through *K*–1. A cluster parallel program typically needs to know the size and rank. We obtain them by calling methods on the world communicator.

```
   static Comm world;
   static int size;
   static int rank;

   public static void main
      (String[] args)
      throws Exception
      {
      Comm.init (args);
      world = Comm.world();
      size = world.size();
      rank = world.rank();
      }
```

Rather than using a loop to execute the computations (subroutine calls) in sequence, we want the processes to execute the computations in parallel. We've already got multiple processes running, so we simply call the `isPrime()` subroutine (without the loop). However, we want each computation to use a different *x* value. To get this to happen, we use the rank to retrieve the appropriate command-line argument.

```
   static Comm world;
   static int size;
   static int rank;
   static long x;

   public static void main
      (String[] args)
      throws Exception
      {
      Comm.init (args);
      world = Comm.world();
      size = world.size();
      rank = world.rank();
      x = Long.parseLong (args[rank]);
      isPrime (x);
      }
```

Here is the complete Java class, Program1Clu, including code to record the running-time measurements and print them after the computation has finished.

```
import edu.rit.pj.Comm;
public class Program1Clu
   {
   static Comm world;
   static int size;
   static int rank;
   static long x;
   static long t1, t2, t3;

   public static void main
      (String[] args)
      throws Exception
      {
      t1 = System.currentTimeMillis();
      Comm.init (args);
      world = Comm.world();
      size = world.size();
      rank = world.rank();
      x = Long.parseLong (args[rank]);
      t2 = System.currentTimeMillis();
      isPrime (x);
      t3 = System.currentTimeMillis();
      System.out.println
         ("rank = "+rank+" call start = "+(t2-t1)+" msec");
      System.out.println
         ("rank = "+rank+" call finish = "+(t3-t1)+" msec");
      }
```

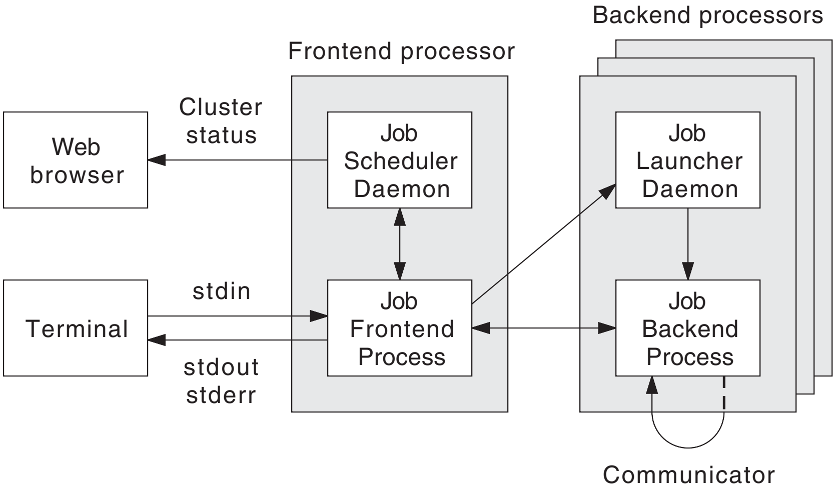


**Figure 19.2** Parallel Java program running on a cluster

```
private static boolean isPrime
   (long x)
   {
   if (x % 2 == 0) return false;
   long p = 3;
   long psqr = p*p;
   while (psqr <= x)
      {
      if (x % p == 0) return false;
      p += 2;
      psqr = p*p;
      }
   return true;
   }
}
```

Here's how the program works. When Parallel Java is installed on a cluster parallel computer, several **daemon processes** are running at all times (Figure 19.2). The **Job Scheduler Daemon** is running on the frontend processor, and a **Job Launcher Daemon** is running on each backend processor. You launch the parallel program on the frontend processor like any other Java program. This creates the **job frontend process** running a JVM. When the program calls the `Comm.init()` method, the job frontend process contacts the Job Scheduler Daemon and asks it to create a new parallel processing job. The Job Scheduler Daemon assigns particular backend processors to the job and informs the job frontend process. The job frontend process then contacts each backend processor's Job Launcher Daemon and asks it to create a new process running a JVM executing the same `main()` method with the same command-line arguments as the job frontend process's JVM. These become the **job backend processes**. The job backend processes establish communication with each other and with the job frontend process.

All of the preceding happens inside the `Comm.init()` method call. In the job frontend process, the `Comm.init()` method does not return; instead, the job frontend process waits for all the job backend processes to finish. In the job backend processes, the `Comm.init()` method call returns, and the main program continues executing. Thus, several `isPrime()` subroutine calls happen at the same time, each subroutine call being performed by a different job backend process with a different argument. Each job backend process prints its own timing measurements on the standard output. The Parallel Java communication layer intercepts these printouts and sends them to the job frontend process, which prints them on its own terminal. When each job backend process finishes executing the `main()` method, it informs the job frontend process. When all the job backend processes have terminated, the job frontend process also terminates.

When running such a process-based program on a cluster parallel computer, the **cluster middleware**—which in Parallel Java consists of the Job Scheduler Daemon, the Job Launcher Daemons, and the communication layers in the job frontend and backend processes—is responsible for creating each process on a different processor. Thus, the computations done by each process—in this case, the different subroutine calls—are executed in parallel on different processors, resulting in a speedup with respect to the sequential program.

The Job Scheduler Daemon also maintains a **job queue** of pending parallel processing jobs. If the backend processors are busy, the job goes into the job queue until the running job finishes and the backend processors become available. In this way, the Job Scheduler Daemon ensures that only one job at a time runs on each backend processor, letting each job utilize the full CPU power of each backend processor. The Job Scheduler Daemon has a Web interface that displays the status of each backend processor and the status of the job queue. Refer to the Parallel Java documentation for further information on installing and configuring Parallel Java on a cluster.

## 19.4 Running the Parallel Program

Here is what Program1Clu printed when run on four processors of the cluster parallel computer. The `-Dpj.np` flag specifies how many processors to use. If this flag is omitted, the default is one processor. The first line of output was printed by the Parallel Java middleware; it gives the job number and the names of the backend processors assigned to the job from rank 0 to rank *K*–1.

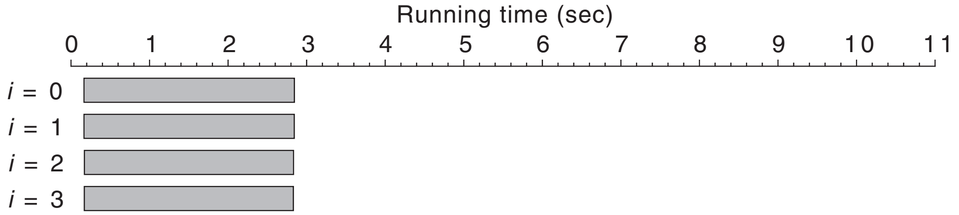


**Figure 19.3** Program1Clu execution timeline, cluster computer

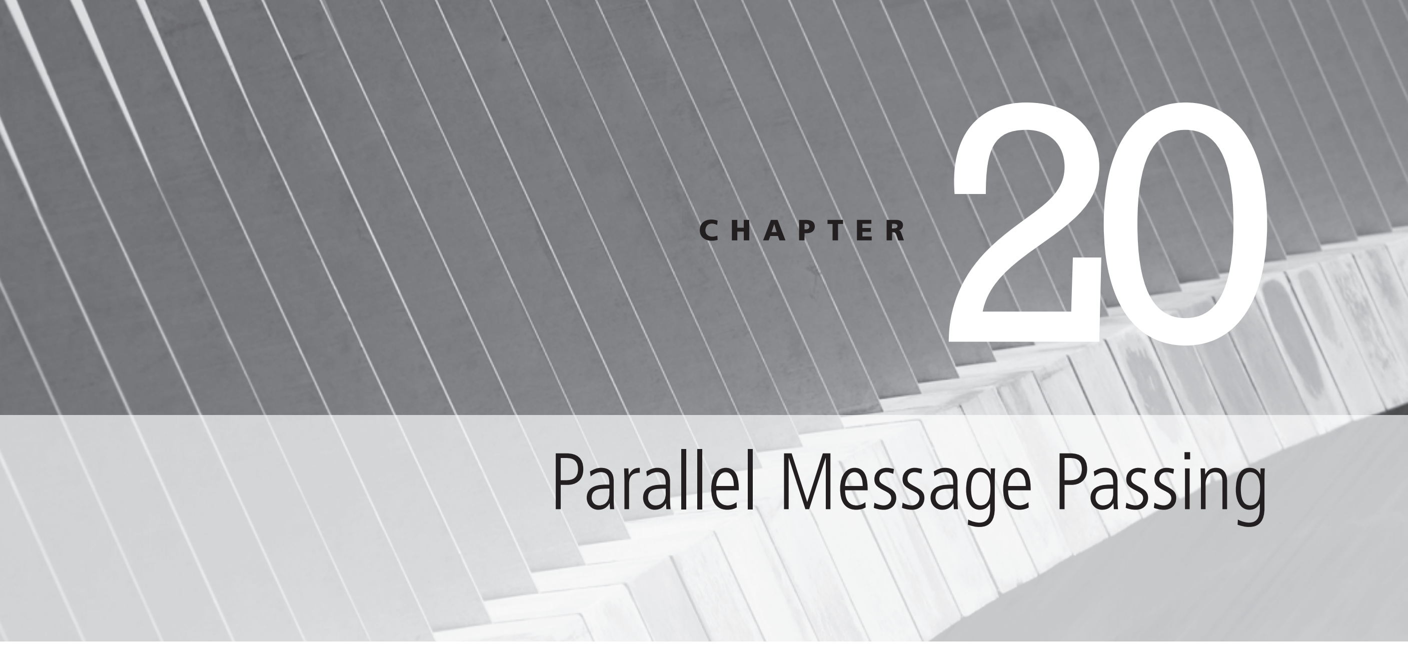
```
$ java -Dpj.np=4 Program1Clu 1000000000000037 1000000000000091 \
  1000000000000159 1000000000000187
Job 3, thug05, thug06, thug07, thug08
rank = 1 call start = 165 msec
rank = 2 call start = 170 msec
rank = 0 call start = 165 msec
rank = 3 call start = 169 msec
rank = 1 call finish = 2842 msec
rank = 2 call finish = 2833 msec
rank = 0 call finish = 2843 msec
rank = 3 call finish = 2830 msec
```

Now the timeline (Figure 19.3) shows parallelism. All the computations start at about the same time, execute simultaneously, and finish at about the same time. Whereas the sequential version's running time was 10611 msec, the parallel version's running time on four processors was 2843 msec—a speedup of 3.732, an efficiency of 0.933. The program's efficiency falls short of ideal because of the time needed to initialize the communication layer (`Comm.init()`).

Note that the processes' printouts do not appear in ascending rank order. This is because the printouts are being done separately by each process running independently of the others, rather than being done all at once by a single process. Doing the printouts in rank order would require some kind of communication or coordination among the processes.

Now that we understand the basics of writing and running programs on a cluster, we can begin our in-depth study of cluster parallel programming techniques. Because cluster parallel programs usually send and receive messages to transfer data between the processes or to coordinate the processes' actions, in Chapter 20, we will start by looking at how message passing works in Parallel Java.

*This page intentionally left blank*



# CHAPTER 20

# Parallel Message Passing

in which we learn how to use a communicator for message passing in a cluster parallel program; and we encounter various patterns for transferring messages between single processes and multiple processes

# CHAPTER 20 Parallel Message Passing

## 20.1 Communicators

In this chapter, we'll introduce Parallel Java's palette of message-passing operations and describe what each one does. Parallel Java's message-passing capabilities are the same as in the MPI standard. (See Appendix B for further information about MPI.) Each message-passing operation is useful in solving certain kinds of problems on cluster parallel computers. In the rest of Part III, we'll study examples of such problems and see how the message-passing operations are used in cluster parallel programs that solve these problems.

As we saw in Chapter 19, a cluster parallel program uses a **communicator** for message passing among the parallel processes. A communicator is an abstraction of the *communication medium* that encompasses a group of processes (Figure 20.1). The communicator's **size** is the number of processes in the group, *K*. Each process has a **rank** in the range 0 through *K*–1; the rank uniquely identifies the process within the communicator.

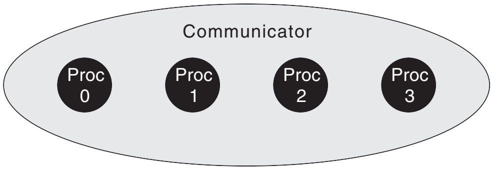


**Figure 20.1** A communicator of size 4

When initialized, the Parallel Java communication layer automatically creates one standard communicator, the **world communicator**. The world communicator encompasses all the processes in the program. The `-Dpj.np` flag on the Java command line specifies the world communicator's size. The program can create additional communicators, if necessary.

In the Parallel Java Library, class edu.rit.pj.Comm provides the API for communicators. Each instance of class Comm represents a different communicator. Class Comm includes these methods:

- The static `Comm.init(args)` method initializes the Parallel Java communication layer. `args` is an array of the program's command-line argument strings.
- The static `Comm.world()` method returns a reference to the world communicator.
- The `size()` method returns the communicator's size.
- The `rank()` method returns the calling process's rank within the communicator.

Typically, the first few lines in a cluster parallel program are these.

```
public static void main
   (String[] args)
   throws Exception
   {
   Comm.init (args);
   Comm world = Comm.world();
   int size = world.size();
   int rank = world.rank();
   . . .
   }
```

At this point, the program has a reference to the world communicator with which to do message-passing operations, the program knows the total number of processes that are participating in the parallel computation, and the program knows the current process's rank within the group.

The other methods of class Comm fall into two categories: point-to-point communication operations; and collective communication operations. For further information about class Comm, refer to the Parallel Java documentation.

## 20.2 Point-to-Point Communication

A **point-to-point communication operation** transfers data from one process to one other process in a communicator. We'll examine six point-to-point communication operations: send, receive, wildcard receive, nonblocking send, nonblocking receive, and send-receive.

**Send and receive**. To send data from one process to another (Figure 20.2), the sending process calls the `send()` method on a communicator, such as the world communicator.

```
world.send (toRank, buf);
```

The first argument specifies the rank of the process to which to send the outgoing data. The second argument is a **buffer**, an instance of class edu.rit.mp.Buf. The buffer object specifies where to obtain the outgoing data item or items. Buffer objects are very flexible; they can refer to a single variable, an array, a portion of an array, a matrix, a portion of a matrix, or other possibilities. We'll discuss the various kinds of buffer objects in Chapter 22. For now, just think of the buffer as an abstract data source.

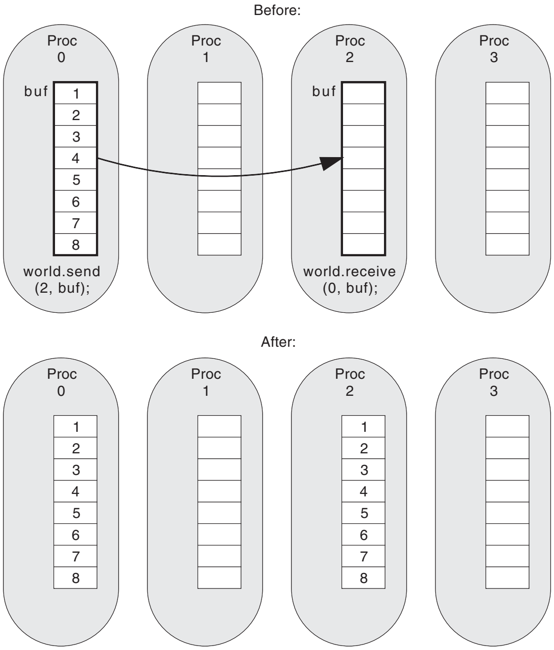


**Figure 20.2** A send operation and a receive operation

To accomplish the data transfer, the receiving process must also call the `receive()` method on the same communicator the sending process is using.

```
    CommStatus status = world.receive (fromRank, buf);
```

The first argument specifies the rank of the process from which to receive the incoming data. The second argument is a buffer object that specifies where to deposit the incoming data item or items. Think of the buffer object in the receive operation as an abstract data destination.

When a process *X* calls `send()` to send data to a process *Y* and process *Y* calls `receive()` to receive data from process *X*, the send operation and the receive operation **match** each other. When a match occurs, a message with the data from process *X*'s source buffer goes through the cluster backend network from process *X* to process *Y*. The message data is then placed into process *Y*'s destination buffer.

In process *Y*, the `receive()` method returns when all the message data has been received. This means that if process *Y* calls `receive()` before process *X* calls `send()`, the `receive()` method call blocks until the data arrives. The `receive()` method also returns a **communication status** object (class edu.rit.pj.CommStatus) that reports the sending process's rank and the number of data items in the message.

In process *X*, the `send()` method returns when all the message data has been sent. However, because of buffering in the underlying network, process *Y* might not have received some or all of the message data when the `send()` method returns in process *X*. Conversely, flow control in the underlying network might cause the `send()` method call to block in process *X* if process *Y* has not called the `receive()` method yet.

Cluster parallel programs that follow the *master-worker* pattern (refer to Section 3.6) use the send and receive operations. The master sends tasks that the workers receive; the workers send results that the master receives. We will see an example of a master-worker program in Chapter 23.

**Wildcard receive**. As just described, the receive operation requires the receiving process to know the rank of the sending process. However, sometimes it's useful for a process to receive a message from *any* sending process, not just one specific sending process. To do so, call the `receive()` method with null as the sending process rank.

```
CommStatus status = world.receive (null, buf);
```

The null acts as a **wildcard** that matches any sending process rank. To discover which process sent a message, examine the sending process rank in the returned communication status object. The master process in a master-worker program typically uses a wildcard receive operation to receive a result from any worker process; we'll see an example in Chapter 23.

**Nonblocking send and receive**. The communicator has *nonblocking* versions of the send and receive operations. The **nonblocking send** method has an additional **communication request** argument.

```
CommRequest request = new CommRequest();
world.send (toRank, buffer, request);
// Other processing goes here
request.waitForFinish();
```

Whereas the `send()` method without a communication request argument blocks the calling thread until the send operation has finished, the `send()` method with a communication request argument initiates the send operation and returns immediately. The message transmission then takes place in a separate thread hidden inside the communication layer. The communication request object (`request`) acts as a "handle" for the in-progress send operation. After returning from the `send()` method, the original thread can perform other processing while the message is being sent. When such other processing is complete and the thread must wait for the send operation to finish before proceeding, the thread calls the communication request object's `waitForFinish()` method; this blocks the calling thread, if necessary, until the send operation has finished.

Likewise, the communicator's **nonblocking receive** method has an additional communication request argument.

```
CommRequest request = new CommRequest();
world.receive (fromRank, buffer, request);
// Other processing goes here
CommStatus status = request.waitForFinish();
```

The nonblocking receive method performs the message reception in a separate thread. To wait for the receive operation to finish, the original thread calls the communication request object's `waitForFinish()` method, which returns a communication status object to report the outcome of the receive operation.

The nonblocking send and receive operations are useful for implementing the *overlapping* pattern in a cluster parallel program. The program initiates a nonblocking send or receive operation; performs some computation (which is thus overlapped with the message I/O); and waits for the send or receive operation to finish. In Chapter 29, we'll study a program that does overlapped computation and communication.

**Send-receive**. The last point-to-point communication operation we examine is **send-receive** (Figure 20.3).

```
CommStatus status =
   world.sendReceive (toRank, srcBuf, fromRank, dstBuf);
```

The calling process both sends data from a source buffer (`srcBuf`) to a process at a given rank (`toRank`) and simultaneously receives data into a destination buffer (`dstBuf`) from a process at a given rank (`fromRank`). The `sendReceive()` method returns a communication status object that reports the status of the receive half of the send-receive operation.

Of course, to accomplish the data transfers, the process at rank `toRank` must perform a receive operation and the process at rank `fromRank` must perform a send operation. If `toRank` and `fromRank` are the same, then that process can also do a send-receive operation, as shown in Figure 20.3. In this way, two processes can exchange data with each other in a single operation. However, `toRank` and `fromRank` need not be the same process (Figure 20.4).

Like the send operation and the receive operation, there is a nonblocking version of the send-receive operation.

```
CommRequest request = new CommRequest();
world.sendReceive (toRank, srcBuf, fromRank, dstBuf, request);
// Other processing goes here
CommStatus status = request.waitForFinish();
```

The send and receive operations are performed in a separate thread. This pattern can be used to overlap computation, sending a message, and receiving a message.

Send-receive operations often turn up in programs where each process repeatedly performs a computation in lockstep with the other processes. The send-receive operations serve both to communicate data among the processes and to synchronize the processes. We'll see examples of such programs in Chapters 28, 29, and 30.

## 20.3 Collective Communication

In contrast to a point-to-point communication operation, which transfers data between only two processes, a **collective communication operation** transfers data among *all* the processes in a communicator. Different collective communication operations transfer data in different patterns; these patterns are often encountered in cluster parallel programs. The collective communication operations are broadcast, flood, scatter, gather, all-gather, reduce, all-reduce, all-to-all, scan, and barrier.

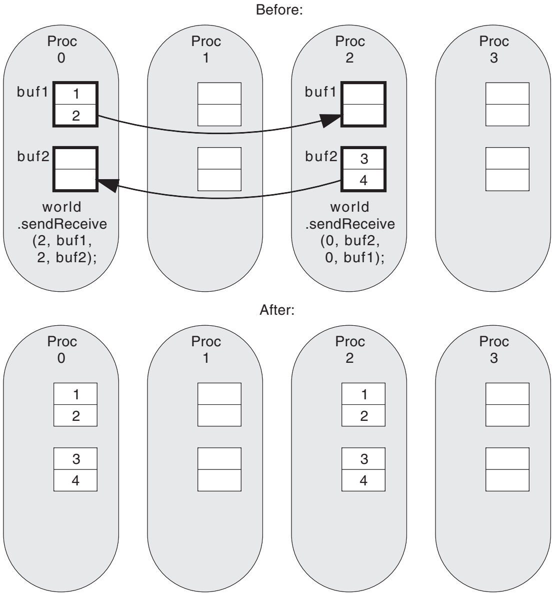


**Figure 20.3** A send-receive operation between two processors

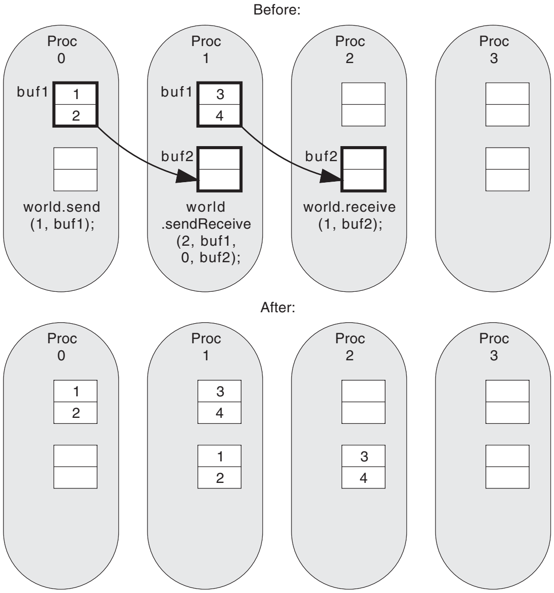


**Figure 20.4** A send-receive operation involving three processors

**Broadcast**. In a broadcast operation (Figure 20.5), one process, called the **root** process, has a buffer of data that it must send to every process. (Any process can be the root.) To accomplish this, every process in the communicator calls the `broadcast()` method.

```
    world.broadcast (root, buf);
```

The first argument is the rank of the root process. In the root process, the second argument is a buffer object specifying the source of the message data. In the non-root processes, the second argument is a buffer object specifying the destination of the message data. The broadcast operation does not take place until *every* process in the communicator has called the `broadcast()` method. After the `broadcast()` method returns, each process's buffer holds the same data as the root process's buffer.

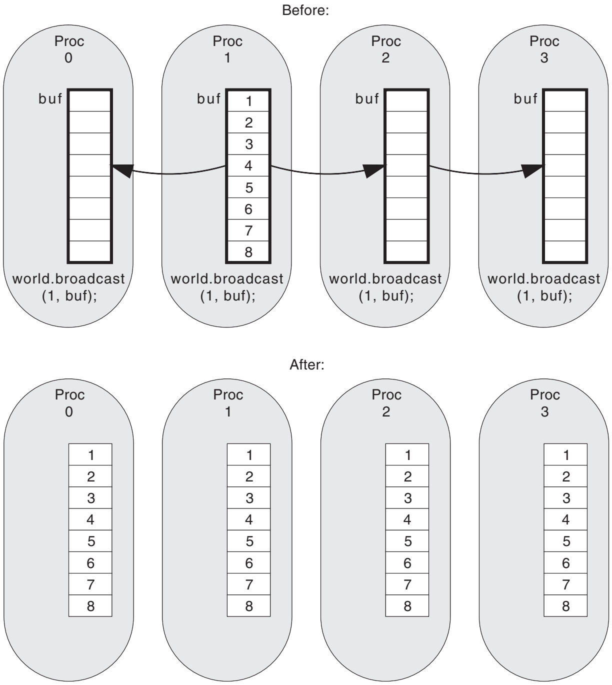


**Figure 20.5** A broadcast operation

The broadcast operation and the other collective communication operations are implemented as a series of point-to-point messages. However, the diagrams in this chapter are intended only to indicate where the data ends up, not the actual messages used to transfer the data. When we examine the performance of message passing starting in Chapter 24, we will look at the actual message patterns for collective communication operations.

On an SMP parallel computer, if every thread must look at a data item, we just put the data in shared memory. That won't work on a cluster parallel computer, which has a distributed memory. One alternative is for the process that owns the data to broadcast the data to all the processes. We'll see an example in Chapter 25.

**Flood**. The flood operation (Figure 20.6) is a combination of the broadcast, send, and receive operations. Like broadcast, the flood operation transfers data from one process to every process in the communicator. Unlike broadcast, the flood operation is split into a flood-send operation and a flood-receive operation. Every process in the communicator calls the `floodReceive()` method, specifying a destination buffer for the incoming message data.

```
    CommStatus status = world.floodReceive (dstBuf);
```

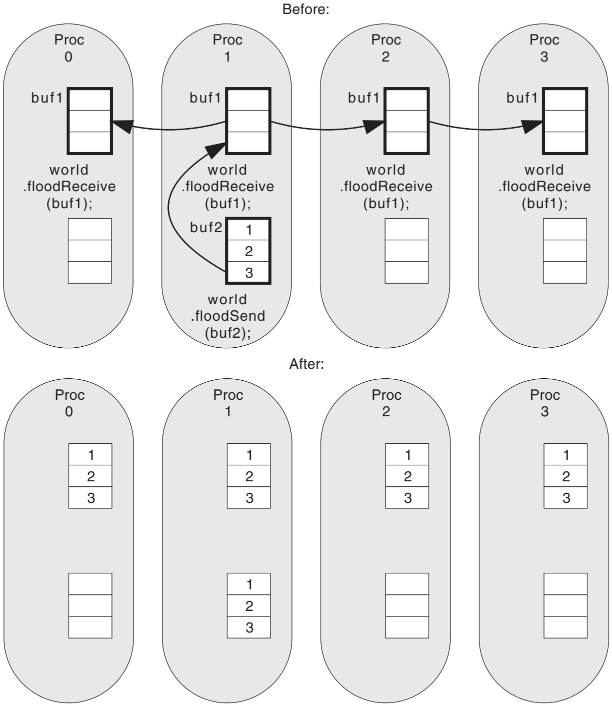


**Figure 20.6** A flood operation

One process in the communicator calls the `floodSend()` method, specifying a source buffer for the outgoing message data.

```
     world.floodSend (srcBuf);
```

The `floodSend()` and `floodReceive()` methods block until every process has called `floodReceive()` and one process has called `floodSend()`. Then the source buffer's contents are "flooded" throughout the communicator to every destination buffer. The `floodReceive()` method returns a communication status object stating which process sent the message and the number of data items in the message.

There are also nonblocking versions of the `floodReceive()` and `floodSend()` methods, similar to the nonblocking `receive()` and `send()` methods.

```
    CommRequest request = new CommRequest;
    world.floodReceive (dstBuf, request);
    // Other processing goes here
    CommStatus status = request.waitForFinish();

    CommRequest request = new CommRequest();
    world.floodSend (srcBuf, request);
    // Other processing goes here
    request.waitForFinish();
```

Flood-send and flood-receive are useful for notifications that must go to all processes, when we don't know ahead of time which process will send the notification. (With a broadcast, the sending process—the root—must be known ahead of time.) In Chapter 21, we'll see a program that uses flood-send and flood-receive to do an early loop exit in all processes.

**Scatter**. In a scatter operation (Figure 20.7), the root process has a group of source data buffers. Every process has one destination data buffer. The source data in the root process is to be distributed, or scattered, among all the processes. To do so, every process calls the `scatter()` method.

```
    world.scatter (root, srcBufArray, dstBuf);
```

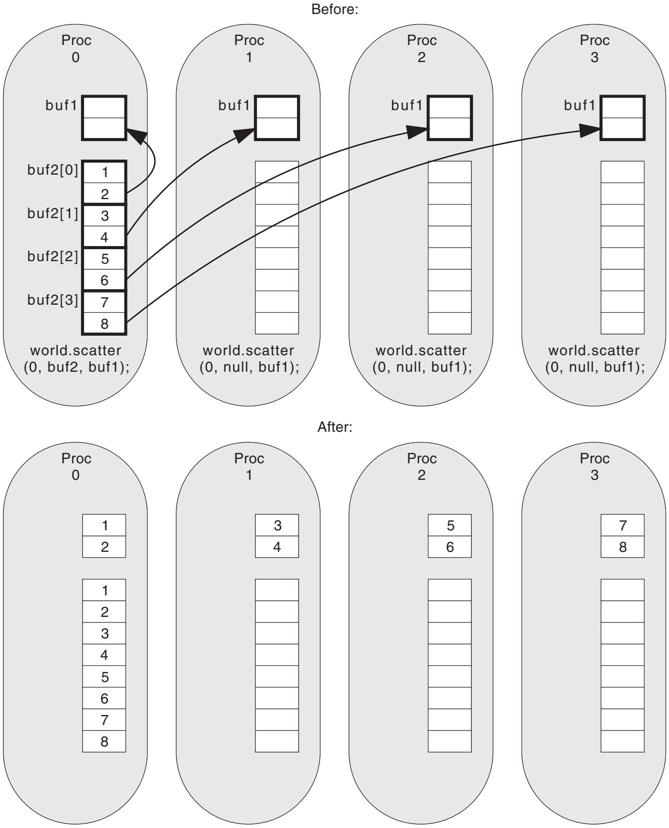

Before:
Proc 0
Proc 1
Proc 2
Proc 3
buf1
buf2[0]
buf2[1]
buf2[2]
buf2[3]
1
2
3
4
5
6
7
8
world.scatter (0, buf2, buf1);
world.scatter (0, null, buf1);
world.scatter (0, null, buf1);
world.scatter (0, null, buf1);
After:
Proc 0
Proc 1
Proc 2
Proc 3
1
2
3
4
5
6
7
8

**Figure** A scatter operation

The first argument is the rank of the root process. In the root process, the second argument is an array of *K* source buffer objects. In the non-root processes, the second argument is not used and can be null. The third argument is the destination buffer object. When all processes have called the `scatter()` method, the contents of the source buffer at index 0 are transferred to the destination buffer in process 0, the contents of the source buffer at index 1 are transferred to the destination buffer in process 1, and so on.

**Gather**. The gather operation (Figure 20.8) is the opposite of the scatter operation. Every process has one source data buffer. The root process has a group of destination data buffers. The source data scattered among all the processes is to be brought back, or gathered, into the root process. To do so, every process calls the `gather()` method.

```
world.gather (root, srcBuf, dstBufArray);
```

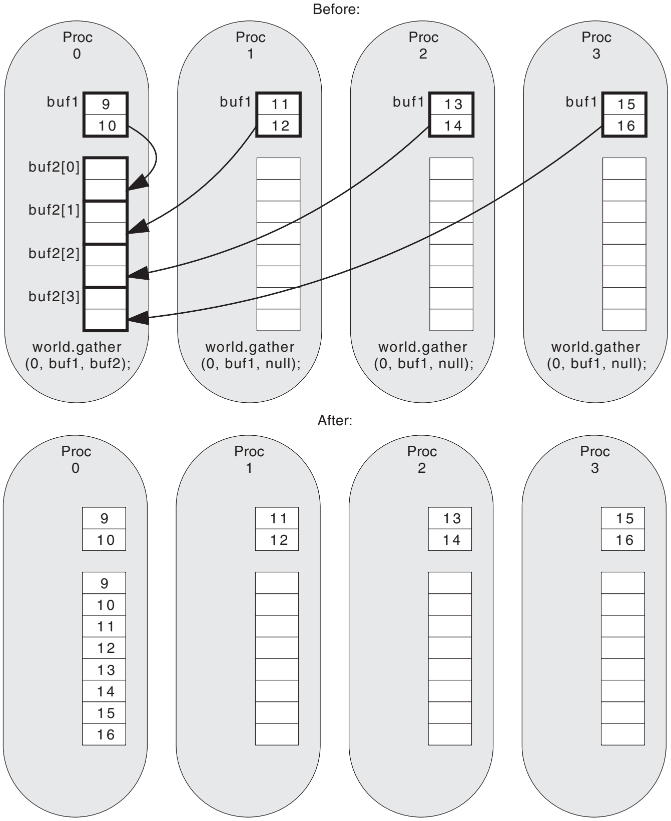


**Figure 20.8** A gather operation

The first argument is the rank of the root process. The second argument is the source buffer object. In the root process, the third argument is an array of *K* destination buffer objects. In the non-root processes, the third argument is not used and can be null. When all processes have called the `gather()` method, the contents of the source buffer in process 0 are transferred to the destination buffer at index 0, the contents of the source buffer in process 1 are transferred to the destination buffer at index 1, and so on.

The scatter and gather operations are often paired in a cluster parallel program. One process, the root process, obtains the program's input data, divides the input data into chunks, and scatters the chunks to all the processes. (Often process 0 acts as the root, although it doesn't matter which process is the root.) Every process performs computations on its own chunk of the input data to produce a chunk of the output data. Finally, the output data chunks are gathered back into the root process. We'll see an example in Chapter 23.

**All-gather**. The all-gather operation (Figure 20.9) is like a gather operation, except that the data is gathered into every process instead of just one process. (The all-gather operation has no root process.) Every process calls the `allGather()` method.

```
    world.allGather (srcBuf, dstBufArray);
```

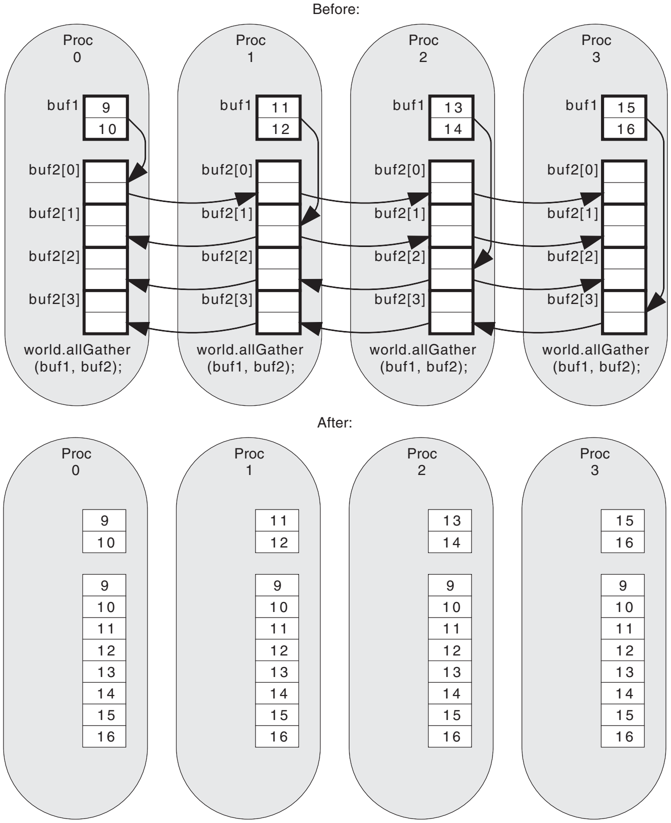


**Figure 20.9** An all-gather operation

The first argument is the source buffer object. The second argument is an array of *K* destination buffer objects. When all processes have called the `allGather()` method, the contents of the source buffer in process 0 are transferred to the destination buffer at index 0 in every process, the contents of the source buffer in process 1 are transferred to the destination buffer at index 1 in every process, and so on.

All-gather operations turn up in programs where each process repeatedly performs a computation in lockstep with the other processes, where the data being computed is partitioned among the processes, and where *every* process must look at *all* the data to compute its own piece of the data. The all-gather operations serve both to keep the data up to date in all the processes and to synchronize the processes. We'll study a program that uses all-gather in Chapter 27.

**Reduce**. The reduce operation (Figure 20.10) does parallel reduction while passing messages. Every process has a data buffer. The many buffers are to be reduced into one using a reduction operator, leaving the result in the root process's buffer. To do so, every process calls the `reduce()` method.

```
    world.reduce (root, buf, op);
```

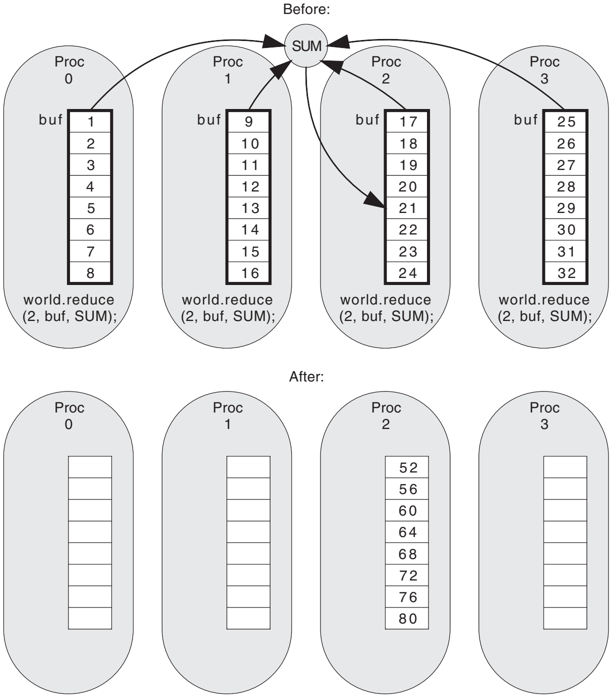


**Figure 20.10** A reduce operation

The first argument is the rank of the root process. The second argument is the data buffer. The third argument is the reduction operator to use, an instance of a class in package edu.rit.pj.reduction. (We studied reduction operators in Chapter 15.) When all processes have called the `reduce()` method, the first data items in all the buffers are combined using the reduction operator and the result becomes the first data item in the root process's buffer; the second data items in all the buffers are combined using the reduction operator and the result becomes the second data item in the root process's buffer; and so on. (In the non-root processes, the buffers are used to hold intermediate results and thus may be altered from their original contents.) In other words, the original data is overwritten.

The reduce operation is used in cluster parallel programs that follow the *parallel reduction* pattern. We'll see some examples in Chapter 26.

**All-reduce**. The all-reduce operation (Figure 20.11) is the same as the reduce operation, except there is no root process, and every process's buffer ends up holding the result of the reduction. Every process calls the `allReduce()` method.

```
    world.allReduce (buf, op);
```

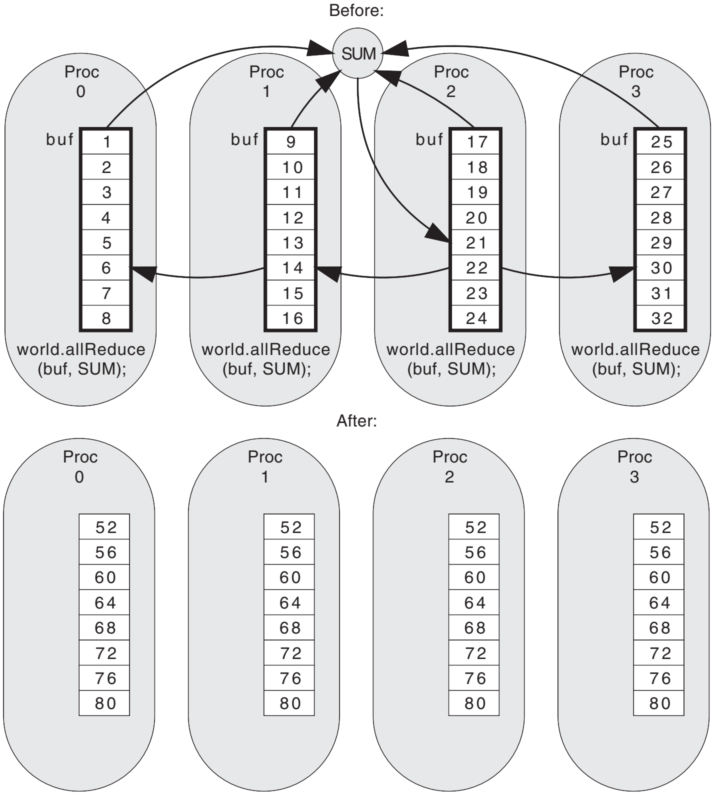


**Figure 20.11** An all-reduce operation

The all-reduce operation is used in cluster parallel programs that follow the *parallel reduction* pattern, but where every process needs to use the overall reduced result. We'll study a program that uses all-reduce in Chapter 30.

**All-to-all**. In a single operation, all-to-all (Figure 20.12) simultaneously scatters a group of source data buffers from every process and gathers a group of destination buffers into every process. To do so, every process calls the `allToAll()` method.

```
    world.allToAll (srcBufArray, dstBufArray);
```

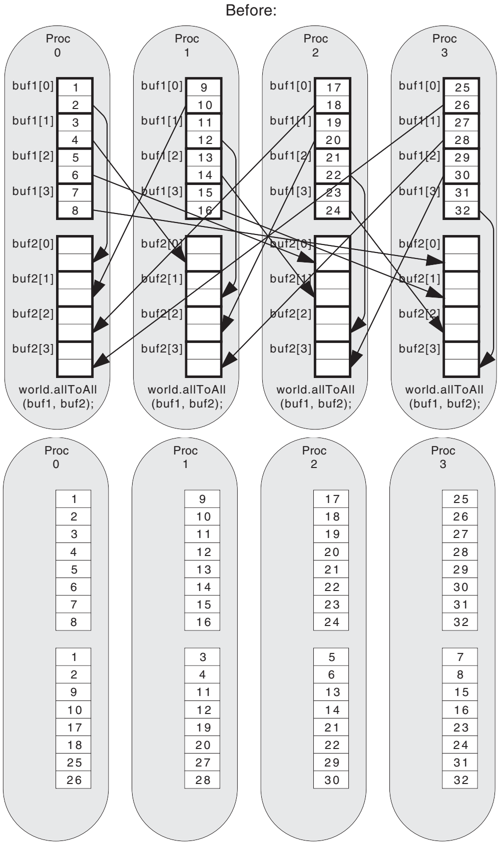


**Figure 20.12** An all-to-all operation

The first argument is an array of *K* source buffer objects. The second argument is an array of *K* destination buffer objects, different from the source buffers. When all processes have called the `allToAll()` method, the source buffers in process 0 are scattered to the destination buffers at index 0 in every process, the source buffers in process 1 are scattered to the destination buffers at index 1 in every process, and so on. Putting it another way, the destination buffers in process 0 are gathered from the source buffers at index 0 in every process, the destination buffers in process 1 are gathered from the source buffers at index 1 in every process, and so on.

Where might all-to-all be useful? One example is a program that does sorting in parallel on a cluster, which we'll study in Chapter 31.

**Scan**. Before explaining the scan operation, consider a simpler operation, **prefix sum**. Applied to an array, prefix sum replaces each array element with the sum of itself and all previous array elements (Figure 20.13).

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Before: | 3 | 56 | 7 | 76 | 22 | 16 | 59 | 94 |
| After: | 3 | 59 | 66 | 142 | 164 | 180 | 239 | 333 |

**Figure 20.13** Prefix sum of an array

The scan operation generalizes the prefix sum operation to use any reduction operator, not just summation. Furthermore, the scan operation uses message passing to combine data items located in buffers in all the processes rather than data elements located in a single array. To do a scan, every process calls the `scan()` method (Figure 20.14).

```
    world.scan (buf, op);
```

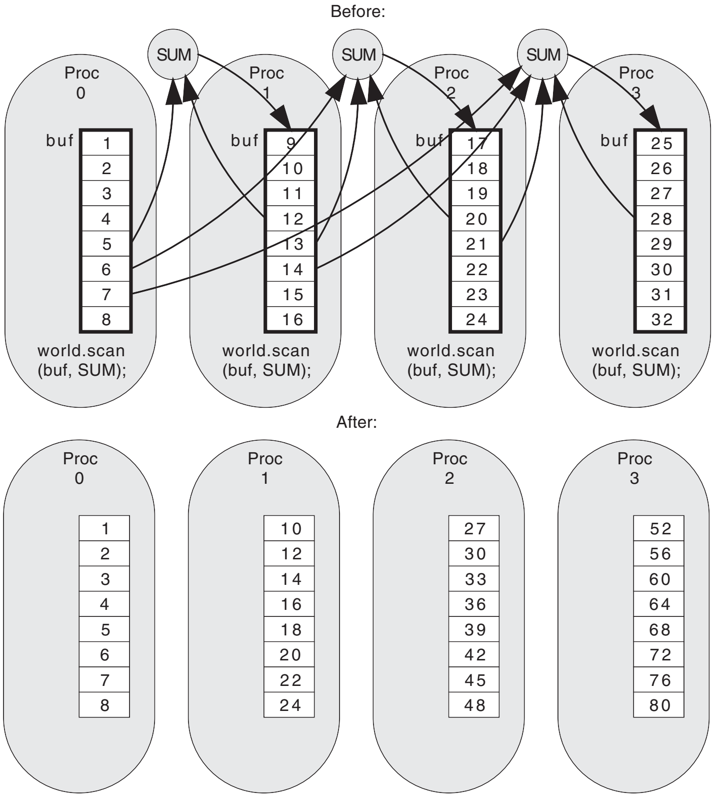


**Figure 20.14** A scan operation

The first argument is a buffer containing the data for the scan. The second argument is the reduction operator to be used. After the scan, the first data item in each process's buffer contains the result of combining the first data items in its own buffer and in all lower-ranked processes' buffers using the reduction operator; the second data item in each process's buffer contains the result of combining the second data items in its own buffer and in all lower-ranked processes' buffers using the reduction operator; and so on.

A variation of prefix sum is **exclusive prefix sum**, which replaces each array element with the sum of all previous array elements, *excluding* itself (Figure 20.15). The first array element is replaced with 0.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Before: | 3 | 56 | 7 | 76 | 22 | 16 | 59 | 94 |
| After: | 0 | 3 | 59 | 66 | 142 | 164 | 180 | 239 |

**Figure 20.15** Exclusive prefix sum of an array

There is also an exclusive-scan operation that generalizes the exclusive prefix-sum operation. To do an exclusive-scan, every process calls the `exclusiveScan()` method (Figure 20.16).

```
    world.exclusiveScan (buf, op, initialValue);
```

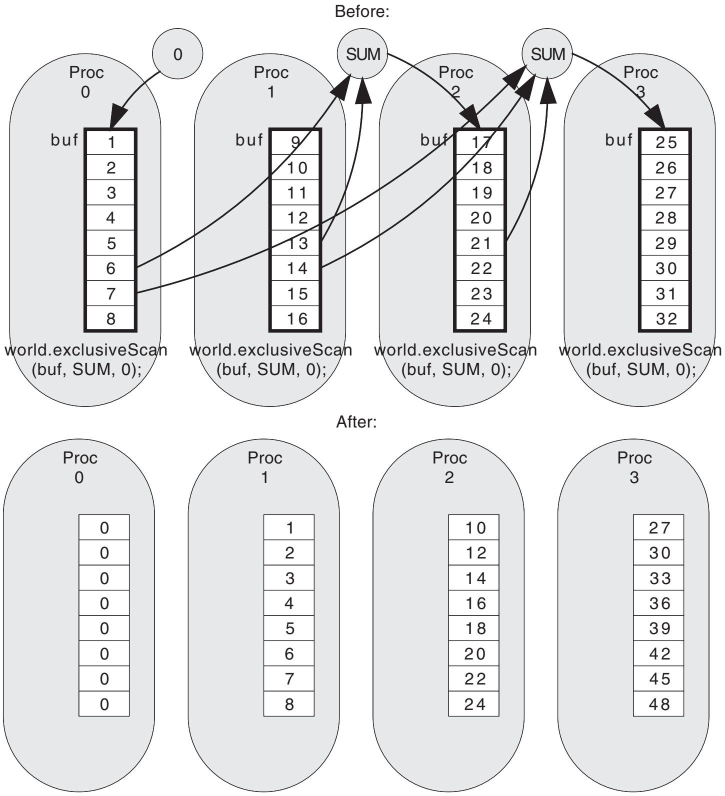


**Figure 20.16** An exclusive-scan operation

The first argument is a buffer containing the data for the scan. The second argument is the reduction operator to be used. The third argument is the initial value to go into the buffer in process 0. After the scan, in process rank 1 and higher, the first data item in each process's buffer contains the result of combining the first data items in all lower-ranked processes' buffers using the reduction operator; the second data item in each process's buffer contains the result of combining the second data items in all lower-ranked processes' buffers using the reduction operator; and so on. In process rank 0, every item in the buffer is set to the initial value.

The program in Chapter 31 that does sorting in parallel on a cluster uses the exclusive-scan operation as well as the all-to-all operation.

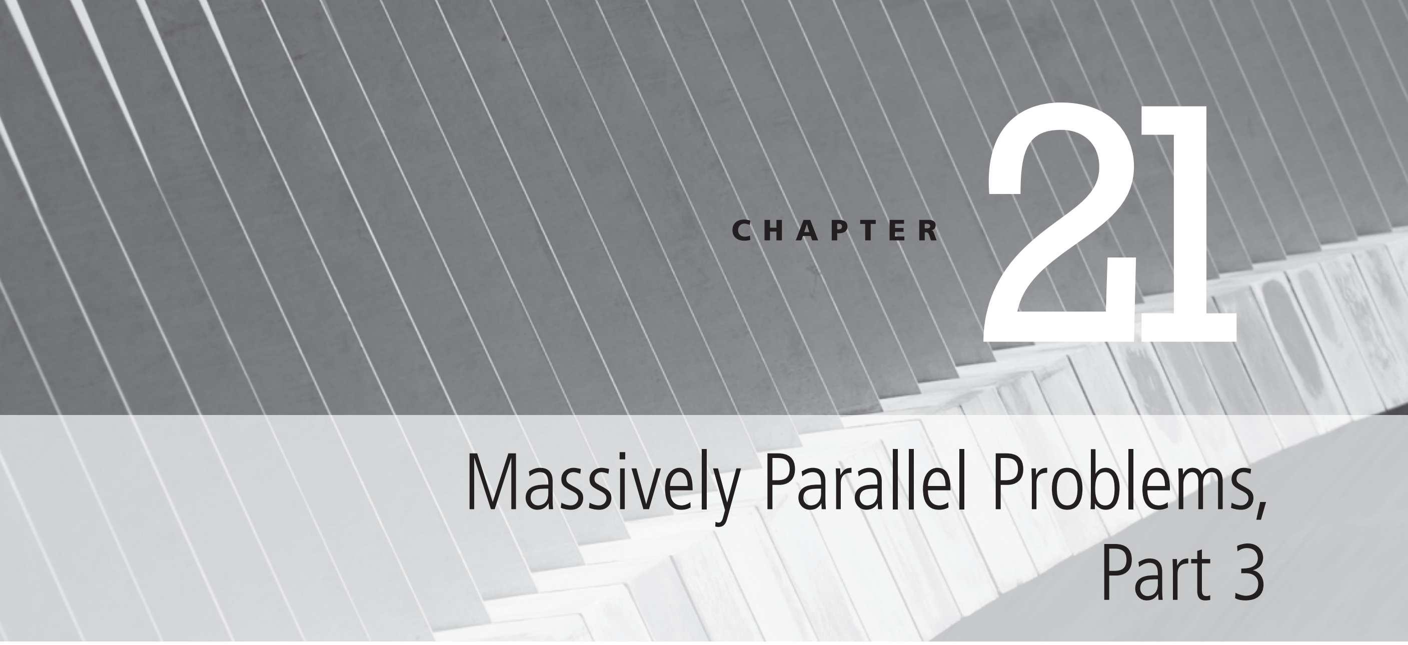
**Barrier**. The final collective communication operation, barrier, uses messages to coordinate all the processes in the communicator, but does not transfer any data. Every process calls the `barrier()` method.

```
    world.barrier();
```

The `barrier()` method call blocks until all processes have called the barrier method, whereupon the `barrier()` method unblocks and returns. This provides a barrier wait among the processes in the communicator, exactly like a barrier wait among the threads in a parallel team that we encountered in the SMP parallel programs in Part II.

Equipped with this palette of communication operations, we are ready to tackle cluster parallel programming in earnest. The remaining chapters in Part III will provide more detailed illustrations of the communication operations and the programming patterns in which they are typically used.

*This page intentionally left blank*



CHAPTER 21

# Massively Parallel Problems, Part 3

in which we convert the sequential program for a massively parallel cryptographic problem into a cluster parallel program; and we begin learning how to apply Parallel Java's cluster parallel programming features

# CHAPTER 21 Massively Parallel Problems, Part 3

## 21.1 Cluster Parallel Program Design

We'll begin our study of cluster parallel programming with the simplest kind of parallel program, a **massively parallel program**. As our example, we'll revisit the AES partial key search program from Chapter 7. Recall that the program takes four command-line arguments: a 128-bit plaintext block, a 128-bit ciphertext block, a 256-bit key with a certain number of low-order bits missing, and $n$, the number of missing key bits. The ciphertext was produced by encrypting the plaintext using the (complete) key. The program's job is to find the correct key by trying to encrypt the plaintext using all $2^n$ possible keys. The program prints the key that successfully reconstructs the ciphertext.

The design considerations for a cluster parallel version of this program are somewhat different from an SMP parallel version. Instead of one process with many threads, each thread running on a different CPU of the same SMP machine, we have many processes with one thread, each process running on a different backend processor of the cluster. Because each process is single-threaded, there are no shared variables accessed by multiple threads, and we don't have to worry about synchronizing the threads. Also, there are no per-thread variables to occupy the same cache lines as other threads' per-thread variables, so we don't have to worry about cache interference. On the other hand, because there are no shared variables, we do have to worry about how the processes will send and receive messages to use data located in other processors. However, in a massively parallel problem, where the pieces are computed independently of each other, there is no need for such communication.

We must divide the computations in the program—the trial encryptions—among the available processors. In the SMP parallel program, the parallel for loop partitioned the computation among the parallel team threads and synchronized the threads to ensure that each thread executed the proper chunk of loop iterations. In the cluster parallel program, there is no need for a central point of coordination like the parallel for loop object; each process needs merely to execute its own chunk of loop iterations.

The sequential key search program does $N = 2^n$ loop iterations, with loop indexes in the range 0 through $N$–1. To divide the computations (loop iterations) among the processes in the cluster parallel program, each process must do a *subset,* or subrange, of the full range. The number of subranges is $K$, the number of processes—the world communicator's *size.* Each process's *rank* within the world communicator determines which subrange the process will do—process 0 will do the first subrange, process 1 will do the second subrange, and so on. An instance of class edu.rit.util.Range represents a range of indexes and has a method to extract a certain subrange from that range.

```
    Range range = new Range (0, N-1);
    Range subrange = range.subrange (size, rank);
```

The Range object represents an index range from *L* through *U* inclusive, where the constructor arguments *L* and *U* are the lower and upper bounds of the complete range. The `subrange()` method divides the range into a number of subranges given by the first argument (`size`) and returns the subrange associated with the second argument (`rank`); each subrange is the same length (plus or minus one). The subrange's lower and upper bounds can then control the process's computation loop.

```
   int lb = subrange.lb();
   int ub = subrange.ub();
   for (int i = lb; i <= ub; ++ i)
      {
      . . .
      }
```

Because each process does a different subrange of the loop iterations, and because together the subranges cover the computation's full range, the cluster parallel program does the same computations as the sequential program. But because all the processes are running at the same time in different processors, the cluster parallel program should experience a speedup or sizeup.

## 21.2 Parallel Key Search Program

Taking the foregoing design considerations into account, here is the code for the cluster parallel version of the AES key search program, FindKeyClu.

```
package edu.rit.clu.keysearch;
import edu.rit.crypto.blockcipher.AES256Cipher;
import edu.rit.pj.Comm;
import edu.rit.util.Hex;
import edu.rit.util.Range;
public class FindKeyClu
   {
   // Command line arguments.
   static byte[] plaintext;
   static byte[] ciphertext;
   static byte[] partialkey;
   static int n;

   // Variables for doing trial encryptions.
   static int keylsbs;
   static int maxcounter;
   static byte[] foundkey;
   static byte[] trialkey;
   static byte[] trialciphertext;
   static AES256Cipher cipher;
   /**
    * AES partial key search main program.
    */
   public static void main
      (String[] args)
      throws Exception
      {
      // Start timing.
      long t1 = System.currentTimeMillis();
```

Here are the standard statements to initialize the communication layer, get the number of processes, and get the current process's rank.

```
      // Initialize PJ middleware.
      Comm.init (args);
      Comm world = Comm.world();
      int size = world.size();
      int rank = world.rank();

      // Parse command line arguments.
      if (args.length != 4) usage();

      plaintext = Hex.toByteArray (args[0]);
      ciphertext = Hex.toByteArray (args[1]);
      partialkey = Hex.toByteArray (args[2]);
      n = Integer.parseInt (args[3]);

      // Make sure n is not too small or too large.
      if (n < 0)
         {
         System.err.println ("n = " + n + " is too small");
         System.exit (1);
         }
      if (n > 30)
         {
         System.err.println ("n = " + n + " is too large");
         System.exit (1);
         }

      // Set up variables for doing trial encryptions.
      keylsbs =
         ((partialkey[28] & 0xFF) << 24) |
         ((partialkey[29] & 0xFF) << 16) |
         ((partialkey[30] & 0xFF) <<  8) |
         ((partialkey[31] & 0xFF)       );
      maxcounter = 1 << n;
      trialkey = new byte [32];
      System.arraycopy (partialkey, 0, trialkey, 0, 32);
      trialciphertext = new byte [16];
      cipher = new AES256Cipher (trialkey);
```

Each process determines the lower and upper bounds of its own loop index subrange, based on the size and rank.

```
      // Determine which chunk of the search space this process
      // will do.
      Range chunk =
         new Range (0, maxcounter-1) .subrange (size, rank);
      int lb = chunk.lb();
      int ub = chunk.ub();
```

And each process only tries encryption keys within that subrange.

```
      // Try every possible combination of low-order key bits.
      for (int counter = lb; counter <= ub; ++ counter)
         {
         // Fill in low-order key bits.

         int lsbs = keylsbs | counter;
         trialkey[28] = (byte) (lsbs >>> 24);
         trialkey[29] = (byte) (lsbs >>> 16);
         trialkey[30] = (byte) (lsbs >>>  8);
         trialkey[31] = (byte) (lsbs       );

         // Try the key.
         cipher.setKey (trialkey);
         cipher.encrypt (plaintext, trialciphertext);

         // If the result equals the ciphertext, we found the key.
         if (match (ciphertext, trialciphertext))
            {
            foundkey = new byte [32];
            System.arraycopy (trialkey, 0, foundkey, 0, 32);
            }
         }
```

The process that found the key prints it. The other processes, failing to find the key, print nothing.

```
      // If we found the key, print it.
      if (foundkey != null)
         {
         System.out.println (Hex.toString (foundkey));
         }

      // Stop timing.
      long t2 = System.currentTimeMillis();
      System.out.println ((t2-t1) + " msec (" + rank + ")");
      }

   /**
    * Returns true if the two byte arrays match.
    */
   private static boolean match
      (byte[] a,
       byte[] b)
      {
      boolean matchsofar = true;
      int n = a.length;
      for (int i = 0; i < n; ++ i)
         {
         matchsofar = matchsofar && a[i] == b[i];
         }
      return matchsofar;
      }
   }
```

And that's it! To go from the sequential program to the cluster parallel program, we added code to initialize the communication layer, we changed the bounds on the computation loop, and we made sure that the correct process printed the answer. The parallel program's structure is the same as the sequential program's.

## 21.3 Parallel Program Speedup

In Part III, we'll measure our cluster parallel programs' running times on a 40-processor hybrid parallel computer named "tardis." Each of this computer's ten backend machines has two 2.6-GHz AMD Opteron 2218 dual-core CPU chips and 8 GB of main memory. The backend machines are connected by a 1-Gbps switched Ethernet. Although the "tardis" parallel computer is a hybrid SMP cluster with four shared-memory processors per backend, in Part III we'll treat this computer as a plain cluster. We will run up to four processes on each backend, but each process will run in its own separate address space with no shared memory. In Part IV, we'll study hybrid parallel programming, and there our programs will achieve parallelism using both message passing and multithreading.

Each process in the parallel program reports its own running time. We will take the program's overall running time to be the largest of the individual processes' running times. Some parallel programs (although not FindKeyClu) are designed so that a certain process does more work than the others. Using the longest process running time ensures that we capture all the program's computation in our measurement.

As we did with the SMP parallel programs in Part II, we'll run the cluster parallel program on several problem sizes. For each problem size, we'll do seven program runs and record the minimum of the running time measurements.

Table 21.1 (at the end of the chapter) gives the running-time measurements in milliseconds for the AES key search program for various problem sizes *N*, as well as the speedups, efficiencies, and *EDSFs* calculated from the running times. Figure 21.1 plots the running-time metrics versus the number of processors *K*. We use *K* = 1, 2, 3, 4, 5, 6, 8, 10, 14, 20, 28, and 40; these are spaced more or less equally on a logarithmic scale.

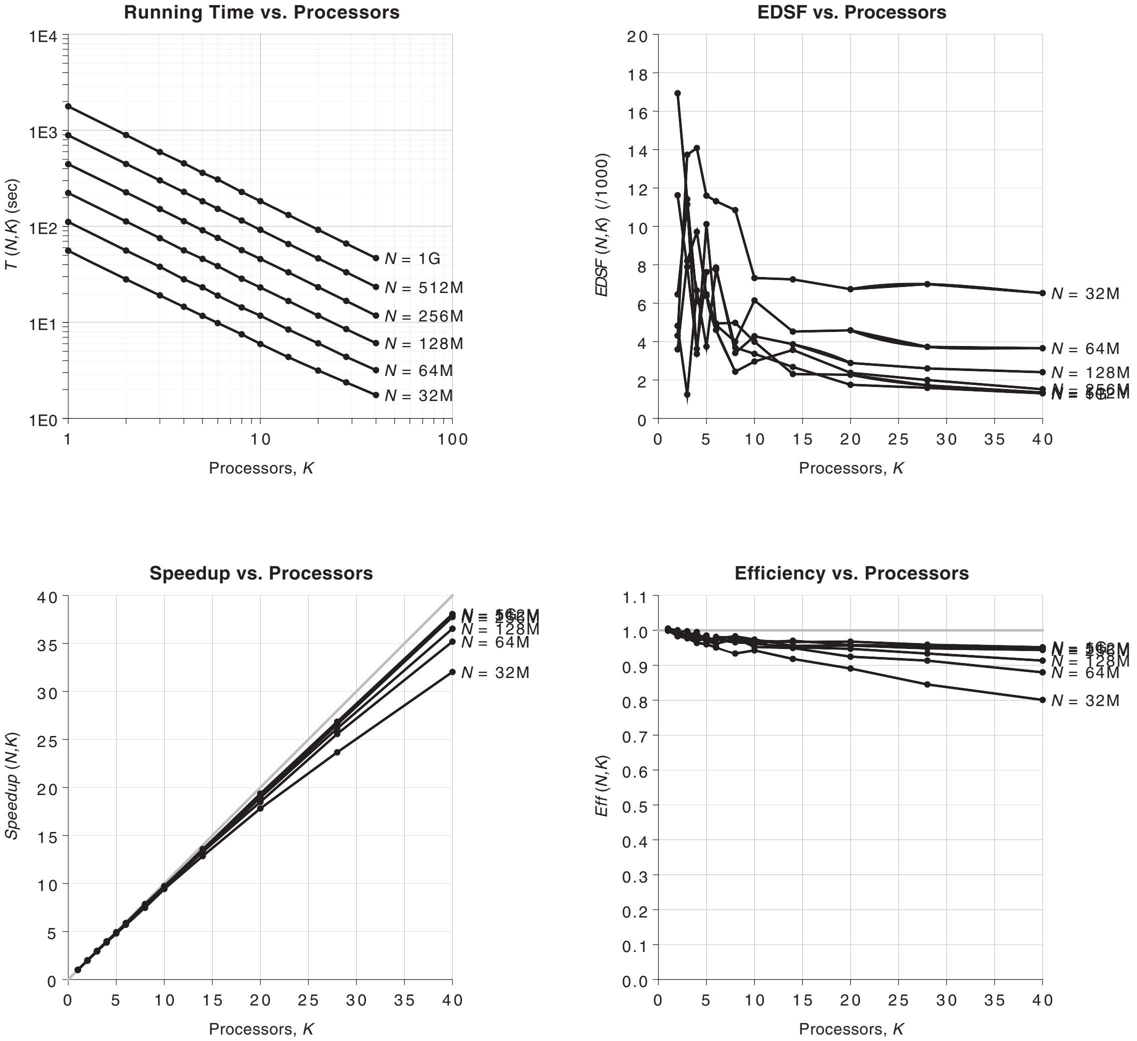


**Figure 21.1** FindKeySeq/FindKeyClu running-time metrics

The speedup and efficiency curves behave as Amdahl's Law predicts, dropping farther away from ideal as the number of processors increases. The cluster parallel program's efficiency drops farther than the SMP parallel program's efficiency did because now we are scaling up to five times as many processors—the cluster has 40 processors, the SMP machine had only 8. Still, with a large problem size, we can achieve excellent efficiencies—95 percent or better for $N = 1\text{G}$ keys searched as $K$ scales up to 40 processors. The high efficiencies are due to low sequential fractions; $F$ is around 2 thousandths for $N = 1\text{G}$.

The *EDSF* curves show that the sequential fraction is more or less constant for each problem size, with fluctuations due to the JIT compiler effect and measurement error for the smaller $K$ values. The sequential fraction decreases as the problem size increases; this suggests that the sequential portion of the running time does not grow as quickly as the total running time. We can confirm this by fitting the running time data to the following model:

$$T(N,K) = (a + bN) + \frac{1}{K}(c + dN) \tag{21.1}$$

The TimeFit program determines that $a = 312$, $b = 5.00 \times 10^{-6}$, $c = 312$, and $d = 1.65 \times 10^{-3}$ msec. For $N$ ranging from 32M to 1G, the sequential portion of the running time ($a + bN$) ranges from 480 to 5,680 msec. While the problem size and total running time grew by a factor of 32, the sequential time grew only by a factor of 12.

## 21.4 Parallel Program Sizeup

Table 21.2 (at the end of the chapter) gives the interpolated problem sizes for selected running times for the AES key search program, as well as the sizeups and sizeup efficiencies. The chosen running times are $T = 10$, 20, 50, 100, 200, and 500 seconds.

Figure 21.2 plots the $N$ versus $K$ data from Table 21.2 for the AES key search program. Each curve is nearly a straight line with unity slope, indicating the ideal situation where the problem size that can be solved in a given amount of time is directly proportional to the number of processors. The plots of *Sizeup* versus $K$ and *SizeupEff* versus $K$ confirm that the program is achieving near-linear sizeups and sizeup efficiencies greater than 95 percent for all the selected running-time values.

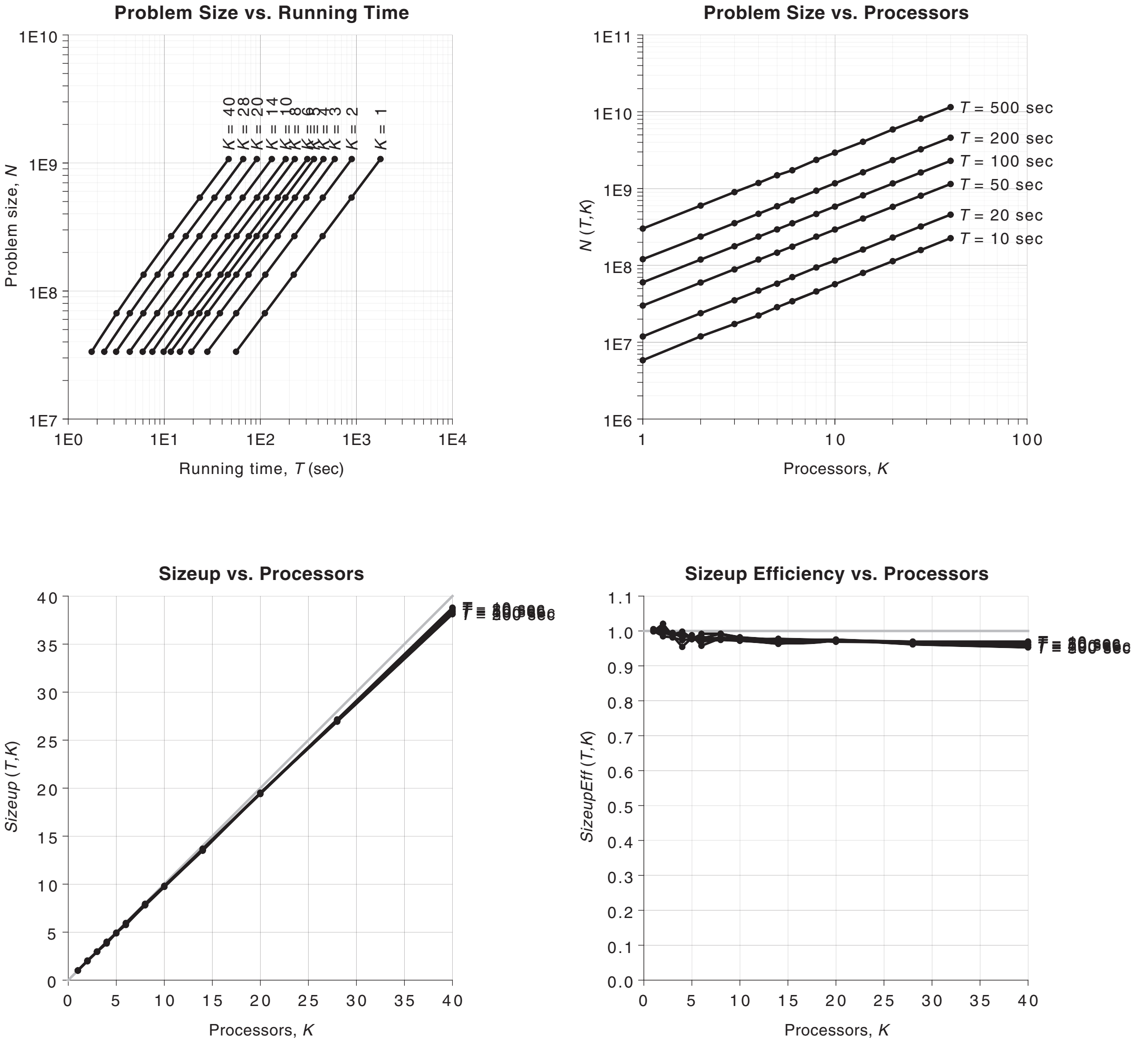


**Figure 21.2** FindKeySeq/FindKeyClu problem-size metrics

The data also illustrate what Gustafson claimed—that when scaling up to more processors, we get better results by increasing the problem size (a sizeup) than by reducing the running time (a speedup). The sizeup and sizeup efficiency curves do not droop as much as the speedup and efficiency curves. The running-time model in Equation 21.1 tells why. For a problem size of $N = 32\text{M}$, for example, the running-time model predicts a sequential fraction $F = 8.54 \times 10^{-3}$. Amdahl's Law then predicts that the maximum speedup is $1/F = 117$. For sizeup, the quantity analogous to $F$ is $G = b/d = 3.03 \times 10^{-3}$. The Second Problem Size Law then predicts that the maximum sizeup is $1 + 1/G = 331$. Figure 21.3 plots these predicted speedups and sizeups as we scale $K$ up to 200 processors. Although both the speedups and the sizeups deviate from the ideal, the sizeups always beat the speedups.

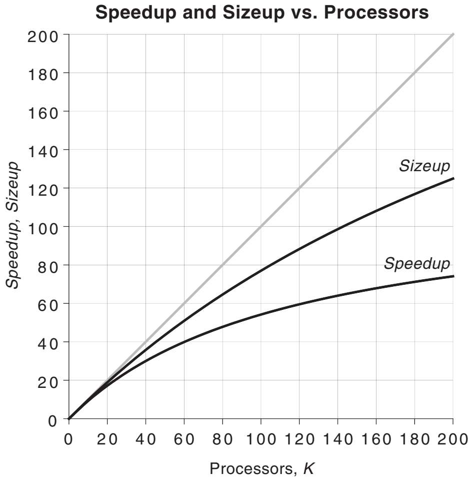


**Figure 21.3** FindKeySeq/FindKeyClu predicted speedup and sizeup

## 21.5 Early Loop Exit

As we did for the SMP version, let's change the cluster version of the AES key search program to stop as soon as it finds the correct key, rather than trying all possible keys. We need *all* the processes to exit their loops as soon as *any* process finds the key. This time, the notification has to use messages instead of a shared variable. When a process finds the key, the process sends a message to every process. When a process receives this message, the process breaks out of its loop. The message doesn't need to include any data; the mere act of receiving the message suffices.

The communicator's *flood* operation is designed precisely for global notifications of this kind. Here is the source code for class FindKeyClu2, which uses a flood operation to trigger an early loop exit.

```
package edu.rit.clu.keysearch;
import edu.rit.crypto.blockcipher.AES256Cipher;
import edu.rit.mp.Buf;
import edu.rit.pj.Comm;
import edu.rit.pj.CommRequest;
import edu.rit.util.Hex;
import edu.rit.util.Range;
public class FindKeyClu2
    {
    // Command line arguments.
    static byte[] plaintext;
    static byte[] ciphertext;
    static byte[] partialkey;
    static int n;
```

```
   // Variables for doing trial encryptions.
   static int keylsbs;
   static int maxcounter;
   static byte[] foundkey;
   static byte[] trialkey;
   static byte[] trialciphertext;
   static AES256Cipher cipher;

   /**
    * AES partial key search main program.
    */
   public static void main
      (String[] args)
      throws Exception
      {
      // Start timing.
      long t1 = System.currentTimeMillis();

      // Initialize PJ middleware.
      Comm.init (args);
      Comm world = Comm.world();
      int size = world.size();
      int rank = world.rank();

      // Parse command line arguments.
      if (args.length != 4) usage();
      plaintext = Hex.toByteArray (args[0]);
      ciphertext = Hex.toByteArray (args[1]);
      partialkey = Hex.toByteArray (args[2]);

      n = Integer.parseInt (args[3]);

      // Make sure n is not too small or too large.
      if (n < 0)
         {
         System.err.println ("n = " + n + " is too small");
         System.exit (1);
         }
      if (n > 30)
         {
         System.err.println ("n = " + n + " is too large");
         System.exit (1);
         }

      // Set up variables for doing trial encryptions.
```

```
keylsbs =
   ((partialkey[28] & 0xFF) << 24) |
   ((partialkey[29] & 0xFF) << 16) |
   ((partialkey[30] & 0xFF) <<  8) |
   ((partialkey[31] & 0xFF)      );
maxcounter = 1 << n;
trialkey = new byte [32];
System.arraycopy (partialkey, 0, trialkey, 0, 32);
trialciphertext = new byte [16];
cipher = new AES256Cipher (trialkey);

// Determine which chunk of the search space this process
// will do.
Range chunk =
   new Range (0, maxcounter-1) .subrange (size, rank);
int lb = chunk.lb();
int ub = chunk.ub();
```

Immediately before beginning the computations, we initiate a *nonblocking* flood-receive operation on the world communicator to receive the notification message, which will arrive sometime in the future. Because the message does not include any data, we specify a zero-length (empty) buffer, returned by the static `IntegerBuf.emptyBuffer()` method, as the destination for the (nonexistent) incoming data. We also specify a communication request object (`req`) with which to keep track of the flood-receive operation's status. The nonblocking `floodReceive()` method returns immediately, allowing the main program thread to proceed to the computations while another thread hidden inside the communication layer waits, poised to receive the message.

```
// Set up to receive a notification when any process finds the
// key.
CommRequest req = new CommRequest();
world.floodReceive (IntegerBuf.emptyBuffer(), req);

// Try every possible combination of low-order key bits.
for (int counter = lb; counter <= ub; ++ counter)
   {
   // Fill in low-order key bits.
   int lsbs = keylsbs | counter;
   trialkey[28] = (byte) (lsbs >>> 24);
   trialkey[29] = (byte) (lsbs >>> 16);
   trialkey[30] = (byte) (lsbs >>>  8);
   trialkey[31] = (byte) (lsbs       );

   // Try the key.
   cipher.setKey (trialkey);
   cipher.encrypt (plaintext, trialciphertext);
```

```
         // If the result equals the ciphertext, we found the key.
         // Send a notification to all processes.
         if (match (ciphertext, trialciphertext))
            {
            foundkey = new byte [32];
            System.arraycopy (trialkey, 0, foundkey, 0, 32);
```

Here is the other half of the flood operation. When we find the key, we do a flood-send operation on the world communicator, specifying a zero-length buffer. A message whose outgoing data comes from the buffer—a message with no data—is sent to every process and is received by the waiting flood-receive operation.

```
            world.floodSend (IntegerBuf.emptyBuffer());
            }
```

Here is where we do the actual early loop exit. At the end of each loop iteration, we check whether the previously initiated communication request (`req`) has finished—whether the flood-receive operation has received a message. If not, we stay in the loop and try the next key; if so, we break out of the loop.

```
         // If key was found, exit loop.
         if (req.isFinished()) break;
         }

      // If we found the key, print it.
      if (foundkey != null)
         {
         System.out.println (Hex.toString (foundkey));
         }

      // Stop timing.
      long t2 = System.currentTimeMillis();
      System.out.println ((t2-t1) + " msec (" + rank + ")");
      }

   /**
    * Returns true if the two byte arrays match.
    */
   private static boolean match
      (byte[] a,
       byte[] b)
      {
      boolean matchsofar = true;
      int n = a.length;
```

```
         for (int i = 0; i < n; ++ i)
            {
            matchsofar = matchsofar && a[i] == b[i];
            }
         return matchsofar;
         }
      }
```

Program FindKeyClu2 illustrates that a cluster parallel program can achieve the same effect with message passing as an SMP parallel program can achieve with a shared variable—in this case, an early loop exit. However, the cluster parallel program requires more machinery to achieve this effect—operations to send and receive messages, buffers to specify the data sources and destinations. In return for the extra effort to write a cluster parallel program, we gain the ability to scale the program up to many more processors and much larger problem sizes than is possible on an SMP parallel computer. We will see this theme played out repeatedly as we build cluster parallel versions of the SMP parallel programs we studied in Part II.

**Table 21.1** FindKeySeq/FindKeyClu running-time metrics

| *N* | *K* | *T* | *Spdup* | *Eff* | *EDSF* | *N* | *K* | *T* | *Spdup* | *Eff* | *EDSF* |
|---:|---:|---:|---:|---:|---:|---:|---:|---:|---:|---:|---:|
| 32M | seq | 56152 | | | | 256M | seq | 445007 | | | |
| 32M | 1 | 55891 | 1.005 | 1.005 | | 256M | 1 | 445115 | 1.000 | 1.000 | |
| 32M | 2 | 28066 | 2.001 | 1.000 | 0.004 | 256M | 2 | 226326 | 1.966 | 0.983 | 0.017 |
| 32M | 3 | 19142 | 2.933 | 0.978 | 0.014 | 256M | 3 | 151760 | 2.932 | 0.977 | 0.011 |
| 32M | 4 | 14563 | 3.856 | 0.964 | 0.014 | 256M | 4 | 113315 | 3.927 | 0.982 | 0.006 |
| 32M | 5 | 11697 | 4.801 | 0.960 | 0.012 | 256M | 5 | 91327 | 4.873 | 0.975 | 0.006 |
| 32M | 6 | 9842 | 5.705 | 0.951 | 0.011 | 256M | 6 | 75895 | 5.863 | 0.977 | 0.005 |
| 32M | 8 | 7517 | 7.470 | 0.934 | 0.011 | 256M | 8 | 56588 | 7.864 | 0.983 | 0.002 |
| 32M | 10 | 5957 | 9.426 | 0.943 | 0.007 | 256M | 10 | 45699 | 9.738 | 0.974 | 0.003 |
| 32M | 14 | 4368 | 12.855 | 0.918 | 0.007 | 256M | 14 | 33268 | 13.376 | 0.955 | 0.004 |
| 32M | 20 | 3152 | 17.815 | 0.891 | 0.007 | 256M | 20 | 23260 | 19.132 | 0.957 | 0.002 |
| 32M | 28 | 2373 | 23.663 | 0.845 | 0.007 | 256M | 28 | 16755 | 26.560 | 0.949 | 0.002 |
| 32M | 40 | 1753 | 32.032 | 0.801 | 0.007 | 256M | 40 | 11787 | 37.754 | 0.944 | 0.002 |
| 64M | seq | 112018 | | | | 512M | seq | 890071 | | | |
| 64M | 1 | 111426 | 1.005 | 1.00 | | 512M | 1 | 890510 | 1.000 | 1.000 | |
| 64M | 2 | 56073 | 1.998 | 0.999 | 0.006 | 512M | 2 | 446856 | 1.992 | 0.996 | 0.004 |
| 64M | 3 | 37970 | 2.950 | 0.983 | 0.011 | 512M | 3 | 301704 | 2.950 | 0.983 | 0.008 |
| 64M | 4 | 28137 | 3.981 | 0.995 | 0.003 | 512M | 4 | 229116 | 3.885 | 0.971 | 0.010 |
| 64M | 5 | 23187 | 4.831 | 0.966 | 0.010 | 512M | 5 | 182643 | 4.873 | 0.975 | 0.006 |
| 64M | 6 | 19023 | 5.889 | 0.981 | 0.005 | 512M | 6 | 152091 | 5.852 | 0.975 | 0.005 |
| 64M | 8 | 14318 | 7.824 | 0.978 | 0.004 | 512M | 8 | 115192 | 7.727 | 0.966 | 0.005 |
| 64M | 10 | 11759 | 9.526 | 0.953 | 0.006 | 512M | 10 | 92254 | 9.648 | 0.965 | 0.004 |
| 64M | 14 | 8427 | 13.293 | 0.949 | 0.005 | 512M | 14 | 65520 | 13.585 | 0.970 | 0.002 |
| 64M | 20 | 6057 | 18.494 | 0.925 | 0.005 | 512M | 20 | 46449 | 19.162 | 0.958 | 0.002 |
| 64M | 28 | 4380 | 25.575 | 0.913 | 0.004 | 512M | 28 | 33282 | 26.743 | 0.955 | 0.002 |
| 64M | 40 | 3183 | 35.193 | 0.880 | 0.004 | 512M | 40 | 23433 | 37.984 | 0.950 | 0.001 |
| 128M | seq | 222581 | | | | 1G | seq | 1780626 | | | |
| 128M | 1 | 222757 | 0.999 | 0.999 | | 1G | 1 | 1780457 | 1.000 | 1.000 | |
| 128M | 2 | 112673 | 1.975 | 0.988 | 0.012 | 1G | 2 | 894524 | 1.991 | 0.995 | 0.005 |
| 128M | 3 | 75424 | 2.951 | 0.984 | 0.008 | 1G | 3 | 594964 | 2.993 | 0.998 | 0.001 |
| 128M | 4 | 56294 | 3.954 | 0.988 | 0.004 | 1G | 4 | 454011 | 3.922 | 0.980 | 0.007 |
| 128M | 5 | 45910 | 4.848 | 0.970 | 0.008 | 1G | 5 | 361425 | 4.927 | 0.985 | 0.004 |
| 128M | 6 | 38585 | 5.769 | 0.961 | 0.008 | 1G | 6 | 308250 | 5.777 | 0.963 | 0.008 |
| 128M | 8 | 28509 | 7.807 | 0.976 | 0.003 | 1G | 8 | 228307 | 7.799 | 0.975 | 0.004 |
| 128M | 10 | 23134 | 9.621 | 0.962 | 0.004 | 1G | 10 | 183436 | 9.707 | 0.971 | 0.003 |
| 128M | 14 | 16710 | 13.320 | 0.951 | 0.004 | 1G | 14 | 131606 | 13.530 | 0.966 | 0.003 |
| 128M | 20 | 11749 | 18.945 | 0.947 | 0.003 | 1G | 20 | 91987 | 19.357 | 0.968 | 0.002 |
| 128M | 28 | 8515 | 26.140 | 0.934 | 0.003 | 1G | 28 | 66321 | 26.849 | 0.959 | 0.002 |
| 128M | 40 | 6092 | 36.537 | 0.913 | 0.002 | 1G | 40 | 46779 | 38.065 | 0.952 | 0.001 |

**Table 21.2** FindKeySeq/FindKeyClu problem-size metrics

| *T* | *K* | *N* | *Sizeup* | *SizeEff* | *T* | *K* | *N* | *Sizeup* | *SizeEff* |
|---:|---:|---:|---:|---:|---:|---:|---:|---:|---:|
| 10000 | seq | 5834457 | | | 100000 | seq | 59890571 | | |
| 10000 | 1 | 5826937 | 0.999 | 0.999 | 100000 | 1 | 60205237 | 1.005 | 1.005 |
| 10000 | 2 | 11910044 | 2.041 | 1.021 | 100000 | 2 | 119191745 | 1.990 | 0.995 |
| 10000 | 3 | 17261962 | 2.959 | 0.986 | 100000 | 3 | 177428466 | 2.963 | 0.988 |
| 10000 | 4 | 22274863 | 3.818 | 0.954 | 100000 | 4 | 237094212 | 3.959 | 0.990 |
| 10000 | 5 | 28598656 | 4.902 | 0.980 | 100000 | 5 | 293930886 | 4.908 | 0.982 |
| 10000 | 6 | 34131885 | 5.850 | 0.975 | 100000 | 6 | 353356405 | 5.900 | 0.983 |
| 10000 | 8 | 45804933 | 7.851 | 0.981 | 100000 | 8 | 467283999 | 7.802 | 0.975 |
| 10000 | 10 | 56936123 | 9.759 | 0.976 | 100000 | 10 | 582478621 | 9.726 | 0.973 |
| 10000 | 14 | 79853310 | 13.687 | 0.978 | 100000 | 14 | 816980300 | 13.641 | 0.974 |
| 10000 | 20 | 113596961 | 19.470 | 0.974 | 100000 | 20 | 1168211204 | 19.506 | 0.975 |
| 10000 | 28 | 158406238 | 27.150 | 0.970 | 100000 | 28 | 1621012487 | 27.066 | 0.967 |
| 10000 | 40 | 226320078 | 38.790 | 0.970 | 100000 | 40 | 2297626293 | 38.364 | 0.959 |
| 20000 | seq | 11840692 | | | 200000 | seq | 120511649 | | |
| 20000 | 1 | 11868970 | 1.002 | 1.002 | 200000 | 1 | 120500107 | 1.000 | 1.000 |
| 20000 | 2 | 23890775 | 2.018 | 1.009 | 200000 | 2 | 237345948 | 1.969 | 0.985 |
| 20000 | 3 | 35083522 | 2.963 | 0.988 | 200000 | 3 | 354796540 | 2.944 | 0.981 |
| 20000 | 4 | 46994497 | 3.969 | 0.992 | 200000 | 4 | 469377827 | 3.895 | 0.974 |
| 20000 | 5 | 57801817 | 4.882 | 0.976 | 200000 | 5 | 588992873 | 4.887 | 0.977 |
| 20000 | 6 | 70460534 | 5.951 | 0.992 | 200000 | 6 | 701580910 | 5.822 | 0.970 |
| 20000 | 8 | 93978892 | 7.937 | 0.992 | 200000 | 8 | 939390015 | 7.795 | 0.974 |
| 20000 | 10 | 115728130 | 9.774 | 0.977 | 200000 | 10 | 1171269075 | 9.719 | 0.972 |
| 20000 | 14 | 160886186 | 13.588 | 0.971 | 200000 | 14 | 1629362518 | 13.520 | 0.966 |
| 20000 | 20 | 230424007 | 19.460 | 0.973 | 200000 | 20 | 2347162655 | 19.477 | 0.974 |
| 20000 | 28 | 321141516 | 27.122 | 0.969 | 200000 | 28 | 3245973630 | 26.935 | 0.962 |
| 20000 | 40 | 457741690 | 38.658 | 0.966 | 200000 | 40 | 4597253261 | 38.148 | 0.954 |
| 50000 | seq | 29859396 | | | 500000 | seq | 301603879 | | |
| 50000 | 1 | 29995070 | 1.005 | 1.005 | 500000 | 1 | 301514139 | 1.000 | 1.000 |
| 50000 | 2 | 59832966 | 2.004 | 1.002 | 500000 | 2 | 600604455 | 1.991 | 0.996 |
| 50000 | 3 | 88663828 | 2.969 | 0.990 | 500000 | 3 | 899891284 | 2.984 | 0.995 |
| 50000 | 4 | 119216727 | 3.993 | 0.998 | 500000 | 4 | 1183527085 | 3.924 | 0.981 |
| 50000 | 5 | 146304623 | 4.900 | 0.980 | 500000 | 5 | 1489873686 | 4.940 | 0.988 |
| 50000 | 6 | 175281662 | 5.870 | 0.978 | 500000 | 6 | 1732973744 | 5.746 | 0.958 |
| 50000 | 8 | 236944791 | 7.935 | 0.992 | 500000 | 8 | 2363261947 | 7.836 | 0.979 |
| 50000 | 10 | 293234960 | 9.821 | 0.982 | 500000 | 10 | 2937640438 | 9.740 | 0.974 |
| 50000 | 14 | 407696961 | 13.654 | 0.975 | 500000 | 14 | 4066509169 | 13.483 | 0.963 |
| 50000 | 20 | 578735478 | 19.382 | 0.969 | 500000 | 20 | 5884017010 | 19.509 | 0.975 |
| 50000 | 28 | 808531916 | 27.078 | 0.967 | 500000 | 28 | 8120857059 | 26.926 | 0.962 |
| 50000 | 40 | 1147812809 | 38.441 | 0.961 | 500000 | 40 | 11496134166 | 38.117 | 0.953 |

# CHAPTER 22

# Data Slicing

in which we see how data gets into and out of messages; and we learn how to slice arrays and dice matrices to partition data among the processes in a cluster parallel program

# CHAPTER 22 Data Slicing

## 22.1 Buffers

In Chapter 20, we examined the point-to-point and collective communication operations, implemented as methods of class Comm. These allow a cluster parallel program to transfer data among the processes in various patterns. At that time, we put off describing how to tell the communication operations where to obtain the outgoing source data and where to deposit the incoming destination data. In Chapter 21, we studied two versions of the cluster parallel AES key search program. One version did not send messages at all. The other version sent a message that contained no data. These programs, however, are atypical; a cluster parallel program usually has to send data between the processes. Before we continue studying cluster parallel programming, we must understand how data gets into and out of messages, and we must understand how a program uses messages to partition data among the processes in the computation.

With Parallel Java, a cluster parallel program uses a **buffer** object to designate a data source or destination. Different abstract base classes provide buffers for each of Java's primitive types as well as for nonprimitive (object) types. These buffer classes, in turn, are subclasses of the abstract base class edu.rit.mp.Buf; objects of type Buf get passed to the communication methods in class Comm. The following table lists the type-specific buffer classes.

| *Java Type* | *Buffer Class* |
|---|---|
| `boolean` | edu.rit.mp.BooleanBuf |
| `byte` | edu.rit.mp.ByteBuf |
| `char` | edu.rit.mp.CharacterBuf |
| `double` | edu.rit.mp.DoubleBuf |
| `float` | edu.rit.mp.FloatBuf |
| `int` | edu.rit.mp.IntegerBuf |
| `long` | edu.rit.mp.LongBuf |
| `short` | edu.rit.mp.ShortBuf |
| `Object` | edu.rit.mp.ObjectBuf |

An IntegerBuf designates a source or destination of `int` data items, a DoubleBuf designates a source or destination of `double` data items, and so on.

An ObjectBuf is for data items of any nonprimitive type. Parallel Java uses **Java Object Serialization** to send and receive objects. When the data items (objects) in an ObjectBuf are sent in an outgoing message, the objects in the buffer are **serialized**, or converted to a sequence of bytes, which are then transmitted. When an incoming message has data items destined for an ObjectBuf, the serialized sequence of bytes is **deserialized**, or converted back to objects, which are then stored in the buffer. For this procedure to work, each object in an ObjectBuf must be **serializable**; its class must implement interface java.io.Serializable and its instance fields must be primitive types or serializable objects.

A program creates a buffer object using a static factory method in the appropriate buffer class. In the rest of the chapter we will look at the factory methods in class IntegerBuf for buffers of `int` data items. The other buffer classes have analogous factory methods; for further information, refer to the Parallel Java documentation. Buffer objects can refer to a single data item, to an array of items, to a portion of an array, to a matrix, or to a portion of a matrix. You can also write your own buffer subclasses to refer to data stored in other kinds of data structures; for further information, refer to the Parallel Java documentation.

## 22.2 Single-Item Buffers

The simplest buffer is one that refers to a single data item (Figure 22.1). To create a single-item buffer, call the `buffer()` method with no arguments:

```
    IntegerItemBuf buf = IntegerBuf.buffer();
```

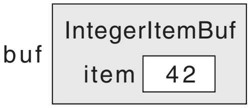


**Figure 22.1** Buffer for a single data item

The buffer object is an instance of class edu.rit.mp.buf.IntegerItemBuf, a subclass of class IntegerBuf. The single data item is located in a public field of the buffer, named `item`. To access the data item in the program, get or set the `item` field:

```
    buf.item = 42;
    System.out.println (buf.item);
```

When this buffer is used as a data source, the outgoing message contains one data item, namely the contents of the `item` field. When this buffer is used as a data destination, the incoming message data is placed in the `item` field.

Many of the cluster parallel programs in the rest of Part III use single-item buffers. In some programs, the items are of primitive types such as `int` and `double`; in other programs, the items are of a nonprimitive (object) type. We'll see examples in Chapters 23, 26, and 30.

Keep in mind that the "single data item" in this kind of buffer can be an object, and thus can actually be an entire data structure—possibly an object with multiple fields of various types, or even a whole collection such as a list or a tree. As long as the object's class is serializable, there will be no difficulty sending and receiving the data item in a message. This is because Parallel Java relies on Java Object Serialization to convert the object to a series of bytes, which are transmitted and used to reconstruct the object on the receiving side. Most of the classes in the Java Collections Framework in package java.util are serializable. As an example utilizing this capability, the cluster parallel program in Chapter 37 uses a single-item ObjectBuf to send a list of “alignments,” where each alignment is itself an object:

```
List<Alignment> alignments = new ArrayList<Alignment>();
world.send (0, ObjectBuf.buffer (alignments));
```

## 22.3 Array Buffers

A buffer can refer to an entire array of data items. A buffer can also refer to a portion, or **slice**, of an array. An array can be partitioned into multiple slices and a buffer created for each slice.

**Buffer for an array**. To create a buffer referring to an entire array (Figure 22.2), call the `buffer()` method, passing in the array:

```
int[] data = new int [8];
IntegerBuf buf = IntegerBuf.buffer (data);
```

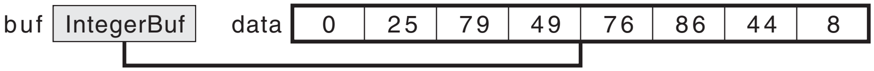


**Figure 22.2** Buffer for an array

In this case, it’s best to view the buffer object as a “handle” that refers to the data items, which are located in the array (`data`). To access the data items in the program, get or set the elements of the array. When this buffer is used as a data source, the outgoing message contains *N* data items, namely all the elements in the array, from index 0 through index *N*–1, where *N* is the array’s length. When this buffer is used as a data destination, the incoming message data items are placed into the array elements from index 0 through index *N*–1.

The cluster parallel programs in Chapters 25, 26, and 31 each use a buffer for an entire array of data. The arrays are communicated among the processes using various message-passing operations, including broadcast, reduce, and exclusive-scan.

**Buffer for an array slice**. To create a buffer for one slice of an array (Figure 22.3), call the `sliceBuffer()` method. The first argument is the array that holds the data. The second argument is a **range** object, an instance of class edu.rit.util.Range, giving the lower and upper array indexes (inclusive) of the array elements to include in the buffer.

```
int[] data = new int [8];
Range slicerange = new Range (2, 4);
IntegerBuf buf = IntegerBuf.sliceBuffer (data, slicerange);
```

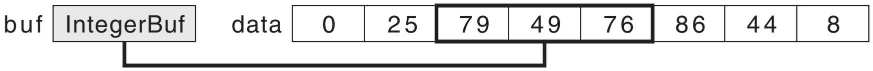


**Figure 22.3** Buffer for an array slice

When this buffer is used as a data source, the outgoing message contains *len* data items, where *len* is the length of the range. These data items come from the array elements at indexes *L* through *U*, where *L* and *U* are the lower and upper bounds of the range. When this buffer is used as a data destination, the incoming message data items are placed into the array elements from index *L* through index *U*.

In a range object, the difference between successive indexes is called the range's **stride**. By default, the stride is 1, and each index in the range is one higher than the previous index. In the preceding example, `slicerange` represents the indexes (2, 3, 4). When used to create a buffer for an array slice, an index range with a stride of 1 yields a buffer that refers to a slice of *contiguous* array elements.

It is also possible to specify a range with a stride greater than 1. To do so, give the stride as the third argument of the range constructor. When used to create a buffer for an array slice, such an index range yields a buffer that refers to a slice of *noncontiguous* array elements. For example, here's how to create a buffer for sending or receiving the array elements at even-numbered indexes (0, 2, 4, 6) in an 8-element array (Figure 22.4).

```
int[] data = new int [8];
Range evenrange = new Range (0, 6, 2);
IntegerBuf buf = IntegerBuf.sliceBuffer (data, evenrange);
```

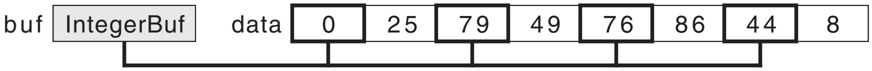


**Figure 22.4** Buffer for array elements at even indexes

Here's how to create a buffer for sending or receiving the array elements at the odd-numbered indexes (1, 3, 5, 7) (Figure 22.5).

```
int[] data = new int [8];
Range oddrange = new Range (1, 7, 2);
IntegerBuf buf = IntegerBuf.sliceBuffer (data, oddrange);
```

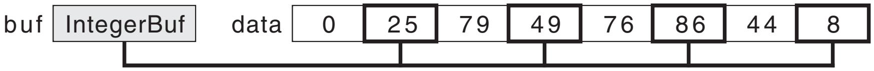


**Figure 22.5** Buffer for array elements at odd indexes

The cluster parallel program in Chapter 30 sends and receives data using buffers referring to slices of noncontiguous array elements, specified by a range with a stride greater than 1.

**Buffers to partition an array**. Suppose we have an array of *N* elements in a parallel program running with *K* processes, and we need to do a computation involving each array element. Often such a program must slice the array and scatter the slices to all the processes, or gather the slices from all the processes, or both. Each array slice is identified by an index range. Together, the *K* index ranges cover all the array indexes from 0 through *N*–1; the slices constitute a **partition** of the array.

To do a scatter or a gather operation, we need an array of buffers, one buffer for each slice of the data array. Here's how to get the array of buffers (Figure 22.6). Create a range object representing the array's complete index range (0 through *N*–1). Call the range object's `subranges()` method, passing in the number of slices, namely the communicator's size *K*. The `subranges()` method partitions the complete range into *K* equal-length subranges and returns an array of range objects representing these subranges. Pass the data array and the array of subranges into the `sliceBuffers()` method. For example, with *N* = 8 and *K* = 4:

```
int[] data = new int [8];
Range[] sliceranges = new Range (0, 7) .subranges (4);
IntegerBuf[] slicebufs =
   IntegerBuf.sliceBuffers (data, sliceranges);
```

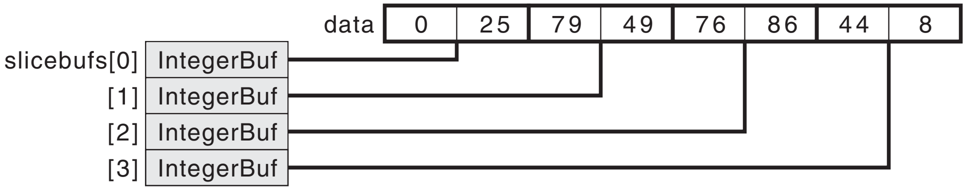


**Figure 22.6** Buffers for an array partition

The `sliceBuffers()` method returns an array of buffer objects. The buffer at index *i* refers to the slice of the data array with index subrange *i*. The array of buffer objects can be used as the source buffer array in a scatter operation; slice *i* of the data array is sent to process *i*. The array of buffer objects can also be used as the destination buffer array in a gather or all-gather operation; slice *i* of the data array is received from process *i*.

The cluster parallel programs in Chapters 27, 28, 29, and 31 partition arrays into equal-sized slices using the `sliceBuffers()` method and transfer the data between processes using various message-passing operations, including all-gather, send-receive, and all-to-all.

## 22.4 Matrix Buffers

A buffer can refer to an entire matrix (two-dimensional array) of data items. A buffer can also refer to a slice of a matrix. A matrix can be partitioned into multiple slices and a buffer created for each slice. With two dimensions, however, there are more ways to slice up a matrix than an array.

**Buffer for a matrix**. To create a buffer referring to an entire matrix (Figure 22.7), call the `buffer()` method, passing in the matrix:

```
int[][] data = new int [4] [8];
IntegerBuf buf = IntegerBuf.buffer (data);
```

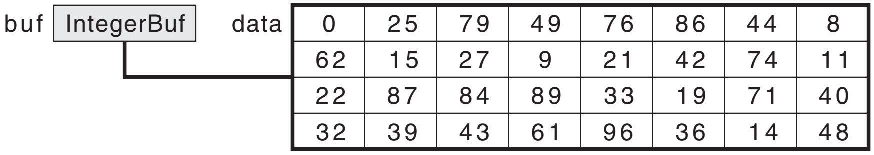
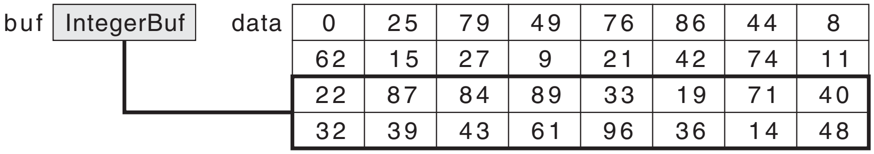


**Figure 22.7** Buffer for a matrix

To access the data items in the program, get or set the elements of the matrix. When this buffer is used as a data source, the outgoing message contains *M*×*N* data items, namely all the elements in the matrix, where *M* is the number of rows and *N* is the number of columns. The data items are extracted from the matrix in **row major order**. That is, the message data consists of the matrix elements in row 0, columns 0 through *N*–1, then the matrix elements in row 1, columns 0 through *N*–1, and so on through row *M*–1. When this buffer is used as a data destination, the incoming message data items are placed into the matrix elements in row major order.

**Buffer for a matrix slice**. A matrix has two dimensions along which we can slice. To slice along the rows (Figure 22.8), creating a **row slice**, call the `rowSliceBuffer()` method. The first argument is the matrix that holds the data. The second argument is a range object giving the lower and upper indexes (inclusive) of the rows to include in the buffer.

```
int[][] data = new int [4] [8];
Range rowrange = new Range (2, 3);
IntegerBuf buf = IntegerBuf.rowSliceBuffer (data, rowrange);
```



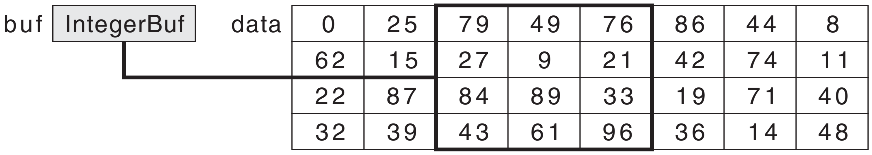
| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 25 | 79 | 49 | 76 | 86 | 44 | 8 |
| 62 | 15 | 27 | 9 | 21 | 42 | 74 | 11 |
| 22 | 87 | 84 | 89 | 33 | 19 | 71 | 40 |
| 32 | 39 | 43 | 61 | 96 | 36 | 14 | 48 |

**Figure 22.8** Buffer for a row slice of a matrix

When this buffer is used as a data source, the outgoing message contains *len*×*N* data items, where *len* is the length of the row range and *N* is the number of columns in the matrix. The data items are extracted from rows *L* through *U* and columns 0 through *N*–1 in row major order, where *L* and *U* are the lower and upper bounds of the row range. When this buffer is used as a data destination, the incoming message data items are placed into rows *L* through *U* and columns 0 through *N*–1 in row major order.

To slice a matrix along the columns (Figure 22.9), creating a **column slice**, call the `colSliceBuffer()` method. The first argument is the matrix that holds the data. The second argument is a range object giving the lower and upper indexes of the columns to include in the buffer.

```
int[][] data = new int [4] [8];
Range colrange = new Range (2, 4);
IntegerBuf buf = IntegerBuf.colSliceBuffer (data, colrange);
```



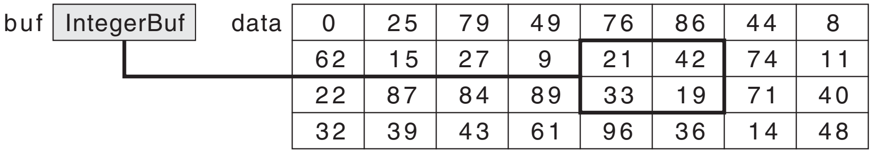
| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 25 | 79 | 49 | 76 | 86 | 44 | 8 |
| 62 | 15 | 27 | 9 | 21 | 42 | 74 | 11 |
| 22 | 87 | 84 | 89 | 33 | 19 | 71 | 40 |
| 32 | 39 | 43 | 61 | 96 | 36 | 14 | 48 |

**Figure 22.9** Buffer for a column slice of a matrix

When this buffer is used as a data source, the outgoing message contains *M*×*len* data items, where *M* is the number of rows in the matrix and *len* is the length of the column range. The data items are extracted from rows 0 through *M*–1 and columns *L* through *U* in row major order, where *L* and *U* are the lower and upper bounds of the column range. When this buffer is used as a data destination, the incoming message data items are placed into rows 0 through *M*–1 and columns *L* through *U* in row major order.

To slice a matrix along both the rows and the columns (Figure 22.10), creating a **patch** of the matrix, call the `patchBuffer()` method. The first argument is the matrix that holds the data. The second argument is a range object giving the lower and upper indexes of the rows to include in the buffer. The third argument is a range object giving the lower and upper indexes of the columns to include in the buffer.

```
int[][] data = new int [4] [8];
Range rowrange = new Range (1, 2);
Range colrange = new Range (4, 5);
IntegerBuf buf =
   IntegerBuf.patchBuffer (data, rowrange, colrange);
```



| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 25 | 79 | 49 | 76 | 86 | 44 | 8 |
| 62 | 15 | 27 | 9 | 21 | 42 | 74 | 11 |
| 22 | 87 | 84 | 89 | 33 | 19 | 71 | 40 |
| 32 | 39 | 43 | 61 | 96 | 36 | 14 | 48 |

**Figure 22.10** Buffer for a patch of a matrix

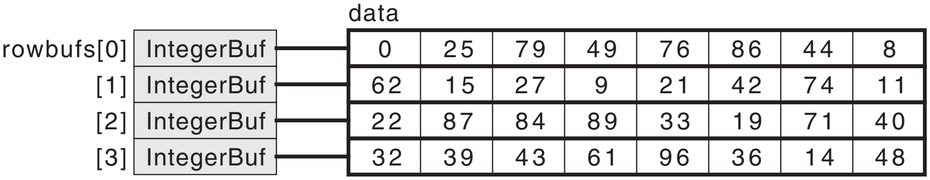
When this buffer is used as a data source, the outgoing message contains *Rlen*×*Clen* data items, where *Rlen* is the length of the row range and *Clen* is the length of the column range. The data items are extracted from rows *RL* through *RU* and columns *CL* through *CU* in row major order, where *RL* and *RU* are the lower and upper bounds of the row range and *CL* and *CU* are the lower and upper bounds of the column range. When this buffer is used as a data destination, the incoming message data items are placed into rows *RL* through *RU* and columns *CL* through *CU* in row major order.

Chapter 23 includes a cluster parallel program that sends row slices of a matrix from one process to another. Because of load balancing, different processes will send matrix row slices of different sizes.

**Buffers to partition a matrix by rows**. As we did with arrays, we can create multiple buffers that partition a matrix. To partition a matrix into row slices (Figure 22.11), call the `rowSliceBuffers()` method. The first argument is the data matrix. The second argument is an array of row index ranges, one for each slice in the partition.

```
int[][] data = new int [4] [8];
Range[] rowranges = new Range (0, 3) .subranges (4);
IntegerBuf[] rowbufs =
   IntegerBuf.rowSliceBuffers (data, rowranges);
```

The `rowSliceBuffers()` method returns an array of buffer objects. The buffer at index *i* refers to the row slice of the data matrix with index subrange *i*. The array of buffer objects can be used as the source buffer array in a scatter operation or the destination buffer array in a gather or all-gather operation.



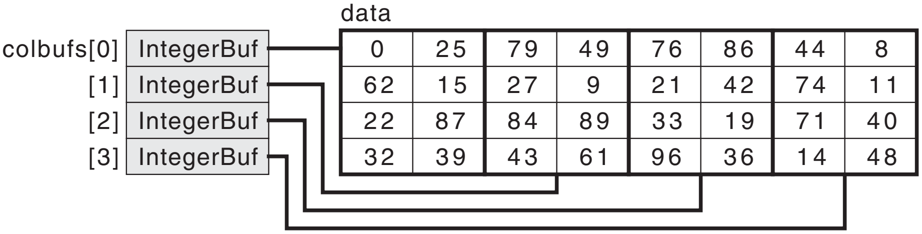
| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 25 | 79 | 49 | 76 | 86 | 44 | 8 |
| 62 | 15 | 27 | 9 | 21 | 42 | 74 | 11 |
| 22 | 87 | 84 | 89 | 33 | 19 | 71 | 40 |
| 32 | 39 | 43 | 61 | 96 | 36 | 14 | 48 |

**Figure 22.11** Buffers for a matrix partitioned by rows

**Buffers to partition a matrix by columns**. To partition a matrix into column slices (Figure 22.12), call the `colSliceBuffers()` method. The first argument is the data matrix. The second argument is an array of column index ranges, one for each slice in the partition.

```
int[][] data = new int [4] [8];
Range[] colranges = new Range (0, 7) .subranges (4);
IntegerBuf[] colbufs =
   IntegerBuf.colSliceBuffers (data, colranges);
```

The `colSliceBuffers()` method returns an array of buffer objects. The buffer at index *i* refers to the column slice of the data matrix with index subrange *i*.



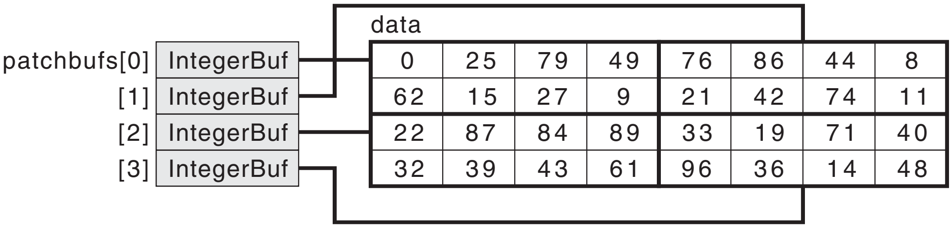
| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 25 | 79 | 49 | 76 | 86 | 44 | 8 |
| 62 | 15 | 27 | 9 | 21 | 42 | 74 | 11 |
| 22 | 87 | 84 | 89 | 33 | 19 | 71 | 40 |
| 32 | 39 | 43 | 61 | 96 | 36 | 14 | 48 |

**Figure 22.12** Buffers for a matrix partitioned by columns

**Buffers to partition a matrix by rows and columns**. To partition a matrix into patches by slicing along both the rows and the columns (Figure 22.13), call the `patchBuffers()` method. The first argument is the data matrix. The second argument is an array of row index ranges. The third argument is an array of column index ranges.

```
int[][] data = new int [4] [8];
Range[] rowranges = new Range (0, 3) .subranges (2);
Range[] colranges = new Range (0, 7) .subranges (2);
IntegerBuf[] patchbufs =
   IntegerBuf.patchBuffers (data, rowranges, colranges);
```

The `patchBuffers()` method returns an array of buffer objects. Each buffer in the array refers to a patch of the matrix with one of the specified row index ranges and one of the specified column index ranges, in every possible combination.



| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 25 | 79 | 49 | 76 | 86 | 44 | 8 |
| 62 | 15 | 27 | 9 | 21 | 42 | 74 | 11 |
| 22 | 87 | 84 | 89 | 33 | 19 | 71 | 40 |
| 32 | 39 | 43 | 61 | 96 | 36 | 14 | 48 |

**Figure 22.13** Buffers for a matrix partitioned into patches

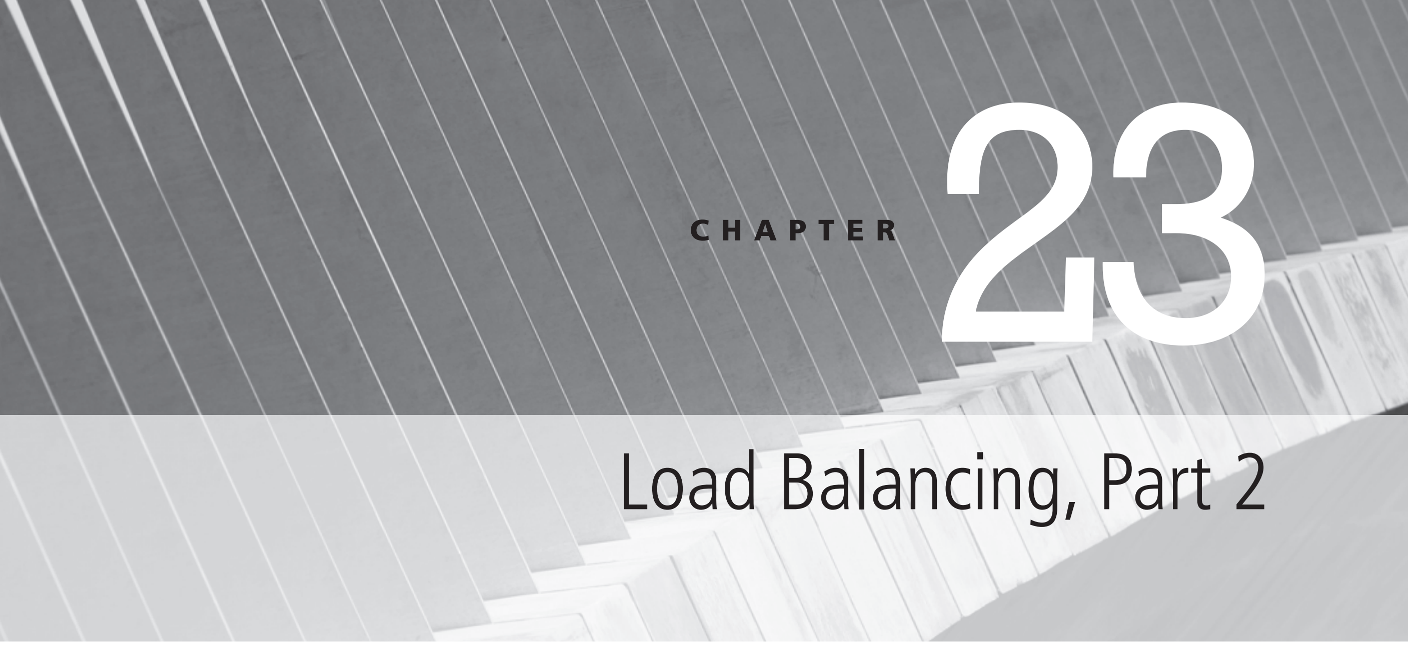
Chapter 23 includes a cluster parallel program that partitions a matrix into equal-sized row slices and gathers the slices from all the processes into one process.

We've now seen most of the methods for creating buffers for single items, arrays, and matrices. Any buffer object can be passed to any communication method. While the number of combinations of buffer objects and communication methods might seem daunting, only a few of the many possible combinations tend to be used in cluster parallel programs. As mentioned earlier, we will see examples of these common patterns in the upcoming chapters.

## 22.5 For Further Information

On Java Object Serialization:

- The Java Tutorials: Basic I/O.
  http://java.sun.com/docs/books/tutorial/essential/io/
- JDK 5.0 Documentation: Object Serialization.
  http://java.sun.com/j2se/1.5.0/docs/guide/serialization/



# CHAPTER 23

# Load Balancing, Part 2

in which we discover that cluster parallel programs can have unbalanced loads just like SMP parallel programs; we learn how to write a cluster parallel program that balances its load; and we see the send, receive, and gather message passing operations in action

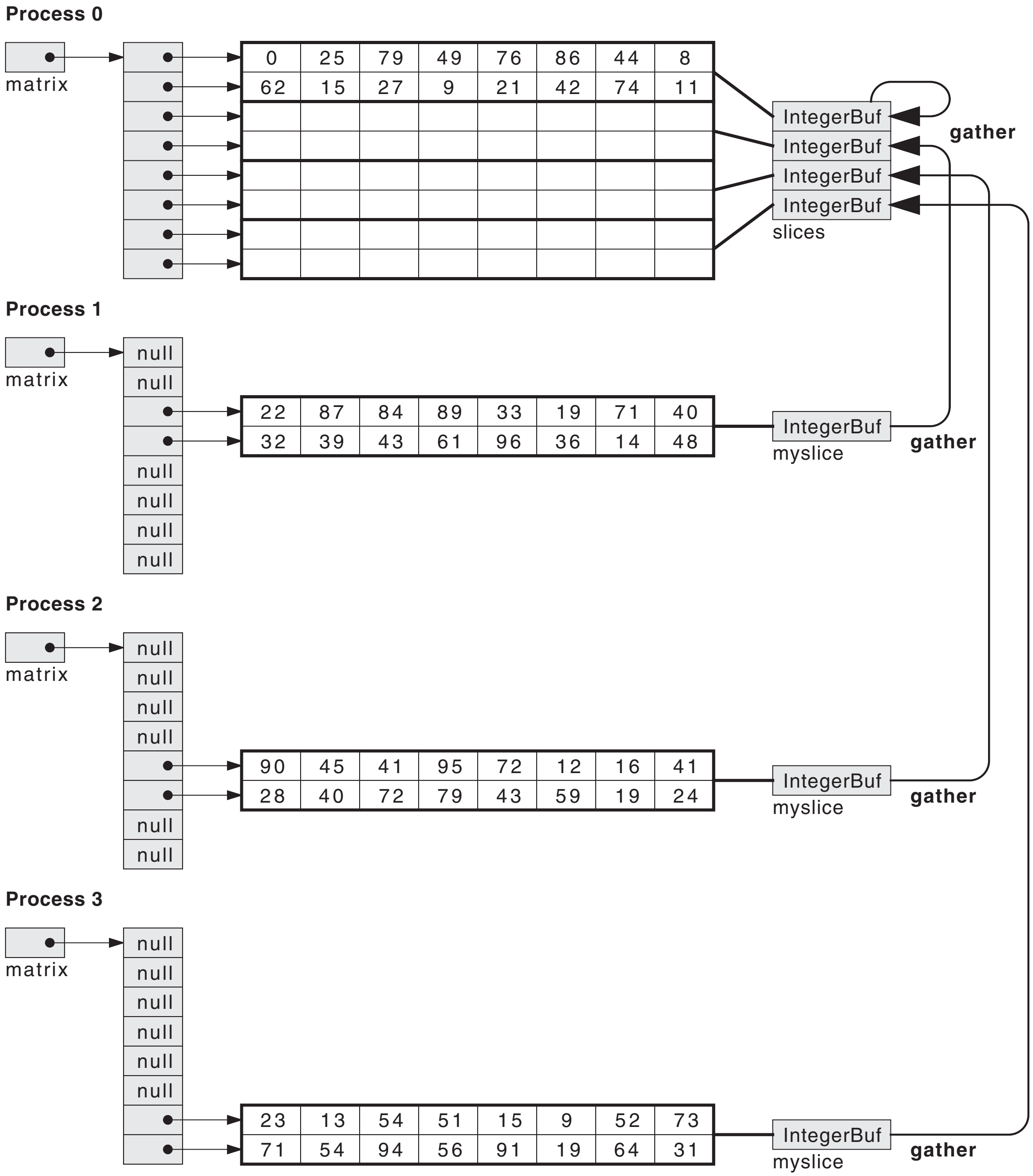
# CHAPTER 23 Load Balancing, Part 2

## 23.1 Collective Communication: Gather

Now that we've seen Parallel Java's message-passing operations and buffer classes, we turn our attention to a cluster parallel program that requires message passing. We'll write a cluster parallel version of the Mandelbrot Set program from Chapter 11. This program computes the color of each pixel in an image of the Mandelbrot Set and stores the pixel data in an integer matrix. The program then writes the pixel data to a PJG image file.

In the SMP parallel version, the pixel data matrix was shared among the threads, and each thread computed a different subset of the shared matrix's rows. We'll use the same concept for the cluster parallel version—each process will compute a different subset of the matrix rows. But in a cluster parallel computer, the pixel data matrix no longer can be a shared variable. Instead, each process must have its own local pixel data matrix, or rather a portion of the matrix. After each process has computed its portion of the matrix, the processes do message passing to bring the matrix portions together into a single process. That process can then write all the pixel data to the PJG image file.

The *gather* collective communication operation can do the job (Figure 23.1). The pixel data matrix is partitioned into *K* row slices, where *K* is the number of parallel processes (the world communicator's size). Process 0 will eventually write the PJG file. The choice of process 0 is arbitrary; any of the processes would serve equally well. Process 0, therefore, allocates storage for the entire matrix and sets up communication buffers for all the matrix slices. Each remaining process allocates storage for, and sets up a communication buffer for, only its own matrix slice (not all the matrix rows). Each process, including process 0, does its own pixel computations and fills in its own slice. All the processes participate in a gather operation to bring the slices together into process 0. Because process 0's slice is already where it needs to be, process 0's part of the gather operation is essentially a no-op. After the gather (Figure 23.2), process 0 writes the now completely filled-in pixel data matrix to the PJG file. Figure 23.3 depicts the parallel program's overall execution timeline, showing the computations and message passing each process performs, and showing process 0 writing the complete image file.



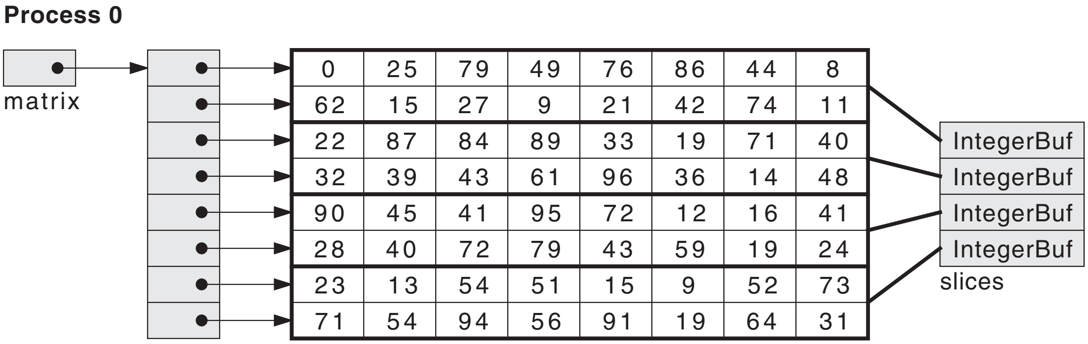
|  |  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|---|
| 0 | 25 | 79 | 49 | 76 | 86 | 44 | 8 |
| 62 | 15 | 27 | 9 | 21 | 42 | 74 | 11 |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |

|  |  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|---|
| 22 | 87 | 84 | 89 | 33 | 19 | 71 | 40 |
| 32 | 39 | 43 | 61 | 96 | 36 | 14 | 48 |

|  |  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|---|
| 90 | 45 | 41 | 95 | 72 | 12 | 16 | 41 |
| 28 | 40 | 72 | 79 | 43 | 59 | 19 | 24 |

|  |  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|---|
| 23 | 13 | 54 | 51 | 15 | 9 | 52 | 73 |
| 71 | 54 | 94 | 56 | 91 | 19 | 64 | 31 |

**Figure 23.1** Pixel data matrix sliced among *K*=4 processes



| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 25 | 79 | 49 | 76 | 86 | 44 | 8 |
| 62 | 15 | 27 | 9 | 21 | 42 | 74 | 11 |
| 22 | 87 | 84 | 89 | 33 | 19 | 71 | 40 |
| 32 | 39 | 43 | 61 | 96 | 36 | 14 | 48 |
| 90 | 45 | 41 | 95 | 72 | 12 | 16 | 41 |
| 28 | 40 | 72 | 79 | 43 | 59 | 19 | 24 |
| 23 | 13 | 54 | 51 | 15 | 9 | 52 | 73 |
| 71 | 54 | 94 | 56 | 91 | 19 | 64 | 31 |

**Figure 23.2** Pixel data matrix in process 0 after the gather

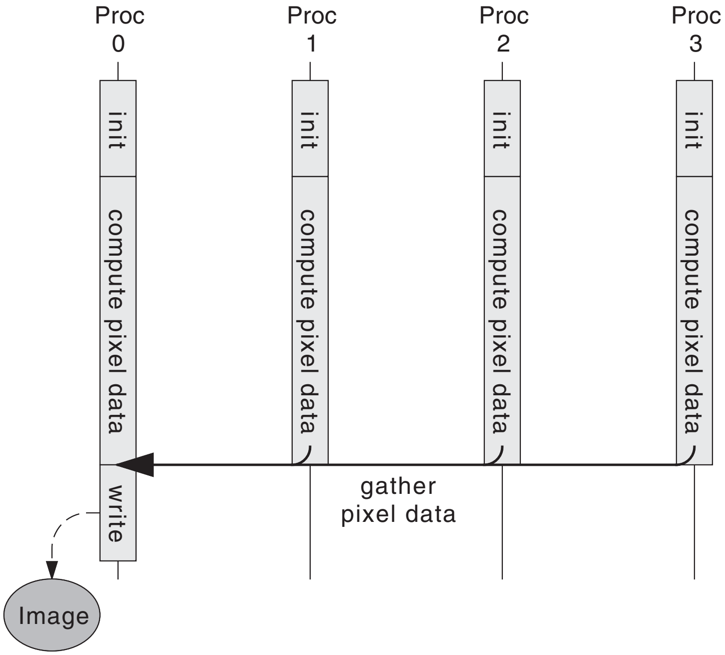


**Figure 23.3** Parallel program execution timeline

## 23.2 Parallel Mandelbrot Set Program

Taking the foregoing design considerations into account, here is the code for the cluster parallel version of the Mandelbrot Set program, MandelbrotSetClu.

```
package edu.rit.clu.fractal;
import edu.rit.color.HSB;
import edu.rit.image.PJGColorImage;
import edu.rit.image.PJGImage;
import edu.rit.mp.IntegerBuf;
import edu.rit.pj.Comm;
import edu.rit.util.Arrays;
import edu.rit.util.Range;
import java.io.BufferedOutputStream;
import java.io.File;
import java.io.FileOutputStream;
public class MandelbrotSetClu
   {
   // Communicator.
   static Comm world;
   static int size;
   static int rank;

   // Command line arguments.
   static int width;
   static int height;
   static double xcenter;
   static double ycenter;
   static double resolution;
   static int maxiter;
   static double gamma;
   static File filename;

   // Initial pixel offsets from center.
   static int xoffset;
   static int yoffset;

   // Image matrix.
   static int[][] matrix;
   static PJGColorImage image;
   static Range[] ranges;
   static Range myrange;
   static int mylb;
   static int myub;

   // Communication buffers.
   static IntegerBuf[] slices;
   static IntegerBuf myslice;

   // Table of hues.
   static int[] huetable;
   /**
    * Mandelbrot Set main program.
    */
   public static void main
      (String[] args)
      throws Exception
      {
      // Start timing.
      long t1 = System.currentTimeMillis();

      // Initialize middleware.
      Comm.init (args);
      world = Comm.world();
      size = world.size();
      rank = world.rank();

      // Validate command line arguments.
      if (args.length != 8) usage();
      width = Integer.parseInt (args[0]);
      height = Integer.parseInt (args[1]);
      xcenter = Double.parseDouble (args[2]);
      ycenter = Double.parseDouble (args[3]);
      resolution = Double.parseDouble (args[4]);
      maxiter = Integer.parseInt (args[5]);
      gamma = Double.parseDouble (args[6]);
      filename = new File (args[7]);

      // Initial pixel offsets from center.
      xoffset = -(width - 1) / 2;
      yoffset = (height - 1) / 2;

      // Create image matrix to store results; the full matrix in
      // process 0, one row slice of the matrix in the other
      // processes.
```

All processes allocate storage for the matrix row references. All processes also partition the matrix into *K* row slices and store the range of row indexes for each slice in the variable `ranges`. Each process's own slice is given by the row range at index `rank`, the process's own rank.

```
      matrix = new int [height] [];
      ranges = new Range (0, height-1) .subranges (size);
      myrange = ranges[rank];
      mylb = myrange.lb();
      myub = myrange.ub();
```

Process 0 allocates storage for all the matrix rows. The other processes allocate storage only for the rows in the process's slice.

```
if (rank == 0)
   {
   Arrays.allocate (matrix, width);
   }
else
   {
   Arrays.allocate (matrix, myrange, width);
   }
```

All processes set up communication buffers for every row slice of the matrix, the row indexes for each slice being specified by the `ranges` variable. Process 0 will use all these buffers (the variable `slices`). The other processes will use only the buffer corresponding to the process's rank (the variable `myslice`).

```
// Set up communication buffers.
slices = IntegerBuf.rowSliceBuffers (matrix, ranges);
myslice = slices[rank];

// Create table of hues for different iteration counts.
huetable = new int [maxiter+1];
for (int i = 0; i < maxiter; ++ i)
   {
   huetable[i] = HSB.pack
      (/*hue*/ (float) Math.pow
         (((double)i) / ((double)maxiter), gamma),
       /*sat*/ 1.0f,
       /*bri*/ 1.0f);
   }
huetable[maxiter] = HSB.pack (1.0f, 1.0f, 0.0f);

long t2 = System.currentTimeMillis();
```

Each process computes the pixel data only for rows in its own slice.

```
// Compute all rows and columns.
for (int r = mylb; r <= myub; ++ r)
   {
   int[] matrix_r = matrix[r];
   double y = ycenter + (yoffset - r) / resolution;
```

```
         for (int c = 0; c < width; ++ c)
            {
            double x = xcenter + (xoffset + c) / resolution;

            // Iterate until convergence.
            int i = 0;
            double aold = 0.0;
            double bold = 0.0;
            double a = 0.0;
            double b = 0.0;
            double zmagsqr = 0.0;
            while (i < maxiter && zmagsqr <= 4.0)
               {
               ++ i;
               a = aold*aold - bold*bold + x;
               b = 2.0*aold*bold + y;
               zmagsqr = a*a + b*b;
               aold = a;
               bold = b;
               }

            // Record number of iterations for pixel.
            matrix_r[c] = huetable[i];
            }
         }

      long t3 = System.currentTimeMillis();
```

After the pixel computations have finished, all processes participate in the gather operation. The destination (root) of the gather is process 0. In every process, the source buffer is the `myslice` variable, which refers just to the process's own row slice. In process 0, the destination buffers are the ones in the `slices` variable, which refer to the slices of the full pixel data matrix.

```
      // Gather all matrix row slices into process 0.
      world.gather (0, myslice, slices);
```

After the gather, process 0 writes the PJG file; the other processes skip this step.

```
      // Write image to PJG file in process 0.
      if (rank == 0)
         {
         image = new PJGColorImage (height, width, matrix);
         PJGImage.Writer writer =
            image.prepareToWrite
               (new BufferedOutputStream
                  (new FileOutputStream (filename)));
         writer.write();
         writer.close();
         }

      // Stop timing.
      long t4 = System.currentTimeMillis();
      System.out.println ((t2-t1) + " msec pre " + rank);
      System.out.println ((t3-t2) + " msec calc " + rank);
      System.out.println ((t4-t3) + " msec post " + rank);
      System.out.println ((t4-t1) + " msec total " + rank);
      }
   }
```

Table 23.1 (at the end of the chapter) lists, and Figure 23.4 plots, the MandelbrotSetClu program's running time data on the "tardis" computer. Each program run calculated the same area as Figure 11.1, with increasing image dimensions $n \times n$ pixels, and with resolutions $r$ increasing in proportion to $n$. Each run's problem size was the total number of pixels, $N = n^2$. The particular $n$, $r$, and $N$ values were the following:

| $n$ | $r$ | $N$ |
|---:|---:|---:|
| 4480 | 1680 | 20M |
| 6400 | 2400 | 40M |
| 8960 | 3360 | 80M |
| 12800 | 4800 | 160M |
| 17920 | 6720 | 320M |
| 25600 | 9600 | 640M |

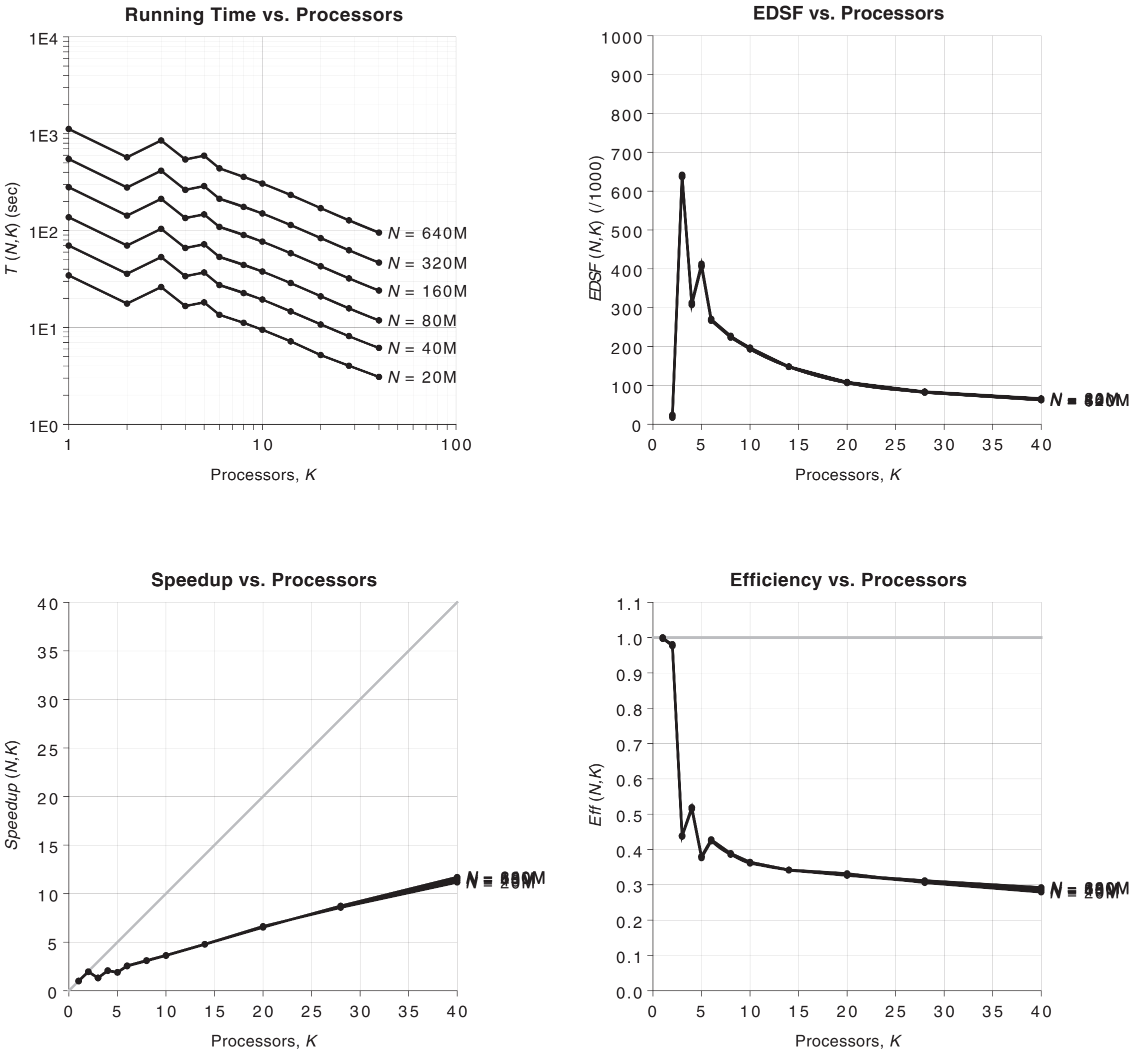


**Figure 23.4** MandelbrotSetSeq/MandelbrotSetClu running-time metrics

The other command-line arguments were *xcenter* = –0.75, *ycenter* = 0, *maxiter* = 1000, and *gamma* = 0.4. The programs were run with *K* = 1, 2, 3, 4, 5, 6, 8, 10, 14, 20, 28, and 40 parallel processes.

Like the SMP parallel version in Chapter 11, the running times fail to diminish in proportion to 1/*K*; the speedups and efficiencies are nowhere near linear. This behavior should come as no surprise. Dividing the pixel data matrix into equal-sized slices in the cluster parallel program is equivalent to using a fixed schedule for the outer parallel for loop in the SMP parallel program. As we have seen, this results in an unbalanced load and poor parallel performance. To get the cluster parallel version to perform properly, we need to balance the load.

## 23.3 Master-Worker

Balancing the load in a cluster parallel program is exactly what the master-worker pattern is designed to do. Figure 23.5 shows the master-worker pattern applied to an agenda parallel problem. To use the master-worker pattern here, we recast the problem of calculating a Mandelbrot Set image, a result parallel problem, as an agenda parallel problem. The tasks in the agenda are to calculate slices of the image; the slices are chosen to yield a balanced load. The master sends these tasks one at a time to the workers. The worker calculates the pixel data for the assigned slice and sends the slice to the master. The master accumulates the slices into its own pixel data matrix. When all the slices have returned from the workers, the master writes the now completely filled-in pixel data matrix to the PJG file.

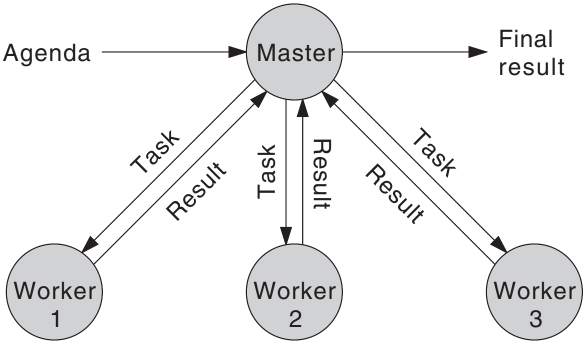


**Figure 23.5** Master-worker pattern

To implement load balancing using the master-worker pattern, we need a way to divide the pixel matrix into row slices such that all the worker processes finish at the same time. In the SMP version, we achieved this by specifying the parallel for loop's schedule as a dynamic or guided schedule. The parallel for loop then divided the outer loop iterations into chunks, each chunk corresponding to a row slice of the pixel data matrix, and handed the slices off to the parallel team threads to be calculated. While we can't use a parallel for loop in the cluster version, we can use the *schedule.* Class edu.rit.pj.IntegerSchedule has methods for dividing a range into chunks. The master process can create an instance of the desired kind of schedule and can call the schedule object's methods to generate agenda items (row slices) for the worker processes.

In a cluster parallel program with *K* processes, one process (process 0, say) can be the master process and *K*–1 processes (processes 1 and higher) can be the worker processes (Figure 23.6). With this design, the master process is blocked most of the time waiting to receive a message from a worker process. When a message arrives, the master process wakes up, does a little flurry of work to store the pixel data and send the next task to the worker process, and then blocks again until the next message arrives.

But if most of the time process 0's CPU is idle because the master process is blocked, then we are not fully utilizing the cluser parallel computer's CPU power. That idle time in process 0 could be put to use calculating pixels. We want to run a worker *thread* in process 0 alongside a separate master *thread;* when the master thread blocks waiting to receive a message, the worker thread can take over the CPU. In other words, in process 0, we want to *overlap* the master and the worker. We'll do it the same way we did overlapped computation and I/O in the continuous cellular automaton program in Chapter 18. A parallel team of two threads executes two parallel sections concurrently, the master section and the worker section.

Now the program has one master thread communicating with one worker thread in the same process and with *K*–1 worker threads in the other processes (Figure 23.7). This is doable; a communicator allows one thread in a process either to send a message to another thread in the same process or to send a message to a different process.

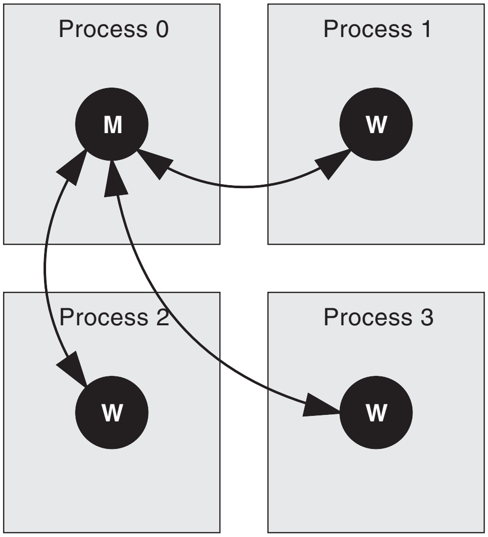


**Figure 23.6** Master-worker program with one master (M) and *K*–1 workers (W)

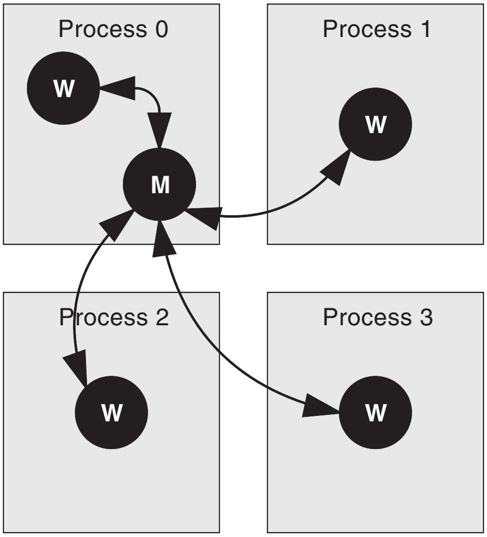


**Figure 23.7** Master-worker program with one master and *K* workers (K)

Figure 23.8 shows the master-worker program's overall execution timeline. Each process initializes its variables. The master thread in process 0 sends the first *K* row slices to the worker threads in processes 0 through *K*–1; each slice is sent as a Range object (lower and upper row indexes). Each worker thread begins computing its respective slice of pixel data. Meanwhile, the master thread waits to receive results from the worker threads. However, the master thread has no way of knowing ahead of time which worker thread will be the first to finish its slice and report the result. Thus, the master thread cannot designate a specific message source rank in the `receive()` method call. Instead, the master must use a *wildcard* to receive a message from any source. Furthermore, the worker thread must first send a message echoing the Range object back to let the master know which chunk of pixel data is about to arrive; the worker thread then sends the actual pixel data in a second message. After storing the second message's contents in the proper rows of the pixel data matrix, the master thread sends the next slice (Range object) to the worker thread, which commences computing the new slice. This continues until there are no more slices. At this point, as each worker thread reports its result, the master thread sends a message containing null (rather than a Range object); this tells the worker thread that there's no more work, whereupon the worker process terminates. After telling every worker process to terminate, the master thread writes the now completely filled-in pixel data matrix to the PJG file.

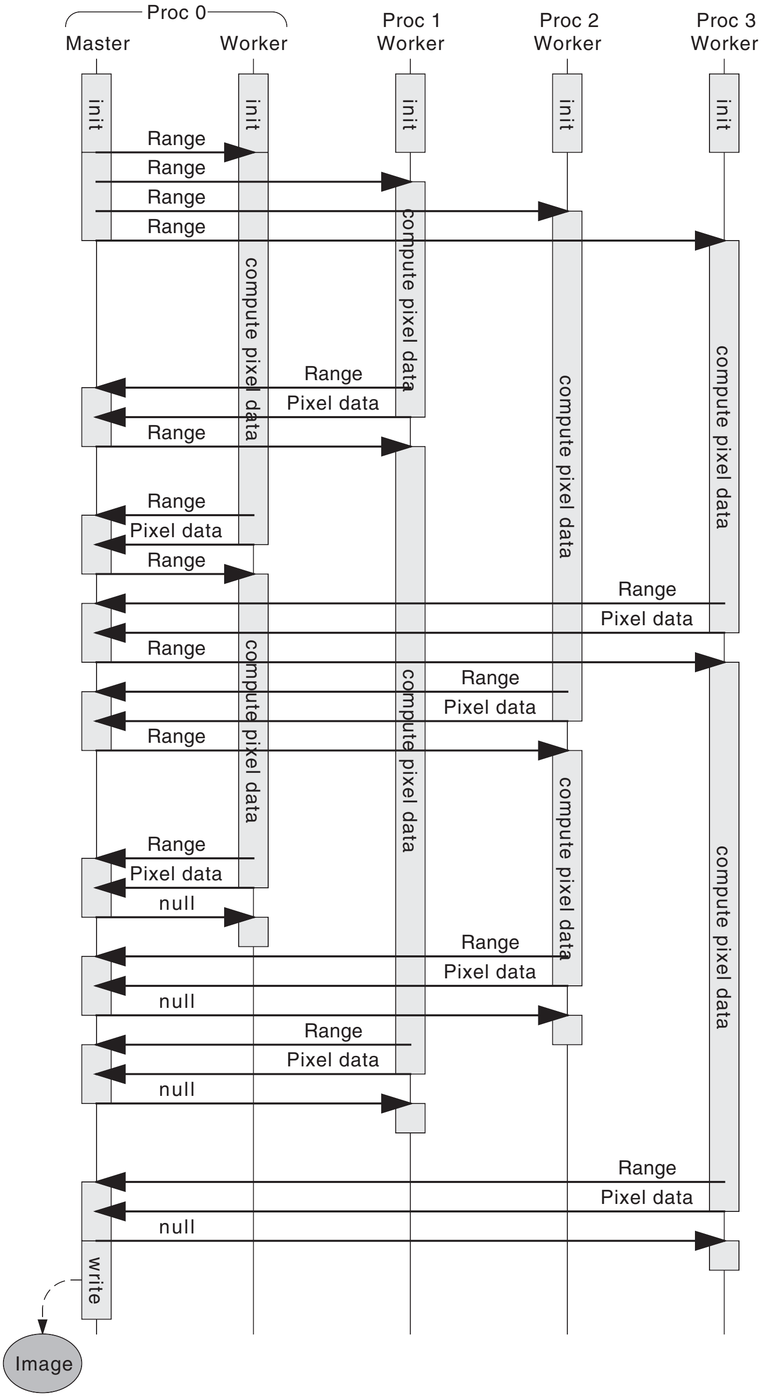


**Figure 23.8** Master-worker program execution timeline

After considering one final detail, we'll be ready to write the code for the master-worker Mandelbrot Set program. Unlike the previous version which had just one kind of message (a gather), the master-worker version has three different kinds of messages: a message sent to a worker containing a range; a message sent to the master containing a range; and a message sent to the master containing pixel data. We need a way to distinguish the various kinds of messages. The communicator's message-passing operations provide **message tags** for this purpose. A tag is just an integer. To send a message with a tag, include the tag after the destination process rank.

```
   world.send (toRank, tag, buffer);
```

To receive a message with a tag, include the tag after the source process rank.

```
   world.receive (fromRank, tag, buffer);
```

If the tag is omitted, it defaults to 0. A `receive()` method call with a tag will match only a `send()` message call with the same tag. By tagging each kind of message with a different tag value, the program can receive the proper kind of message at the proper time. We will use tags to make sure the master thread first receives a message with a Range object, and then receives a message with pixel data.

## 23.4 Master-Worker Mandelbrot Set Program

Here is the code for the second cluster parallel version of the Mandelbrot Set program, MandelbrotSetClu2, which uses the master-worker pattern for load balancing.

```
package edu.rit.clu.fractal;
import edu.rit.color.HSB;
import edu.rit.image.PJGColorImage;
import edu.rit.image.PJGImage;
import edu.rit.mp.IntegerBuf;
import edu.rit.mp.ObjectBuf;
import edu.rit.mp.buf.ObjectItemBuf;
import edu.rit.pj.Comm;
import edu.rit.pj.CommStatus;
import edu.rit.pj.IntegerSchedule;
import edu.rit.pj.ParallelRegion;
import edu.rit.pj.ParallelSection;
import edu.rit.pj.ParallelTeam;
import edu.rit.util.Range;
import java.io.BufferedOutputStream;
import java.io.File;
import java.io.FileOutputStream;
import java.io.IOException;
public class MandelbrotSetClu2
```

```
{
// Communicator.
static Comm world;
static int size;
static int rank;

// Command line arguments.
static int width;
static int height;
static double xcenter;
static double ycenter;
static double resolution;
static int maxiter;
static double gamma;
static File filename;

// Initial pixel offsets from center.
static int xoffset;
static int yoffset;

// Image matrix.
static int[][] matrix;
static PJGColorImage image;

// Table of hues.
static int[] huetable;

// Message tags.
static final int WORKER_MSG = 0;
static final int MASTER_MSG = 1;
static final int PIXEL_DATA_MSG = 2;

// Number of chunks the worker computed.
static int chunkCount;

/**
 * Mandelbrot Set main program.
 */
public static void main
   (String[] args)
   throws Exception
   {
   // Start timing.
   long t1 = System.currentTimeMillis();
```

```
      // Initialize middleware.
      Comm.init (args);
      world = Comm.world();
      size = world.size();
      rank = world.rank();

      // Validate command line arguments.
      if (args.length != 8) usage();
      width = Integer.parseInt (args[0]);
      height = Integer.parseInt (args[1]);
      xcenter = Double.parseDouble (args[2]);
      ycenter = Double.parseDouble (args[3]);
      resolution = Double.parseDouble (args[4]);
      maxiter = Integer.parseInt (args[5]);
      gamma = Double.parseDouble (args[6]);
      filename = new File (args[7]);

      // Initial pixel offsets from center.
      xoffset = -(width - 1) / 2;
      yoffset = (height - 1) / 2;

      // Create table of hues for different iteration counts.
      huetable = new int [maxiter+1];
      for (int i = 0; i < maxiter; ++ i)
         {
         huetable[i] = HSB.pack
            (/*hue*/ (float)
               Math.pow (((double)i)/((double)maxiter),gamma),
             /*sat*/ 1.0f,
             /*bri*/ 1.0f);
         }
      huetable[maxiter] = HSB.pack (1.0f, 1.0f, 0.0f);

      long t2 = System.currentTimeMillis();
```

Here is the parallel team of two threads executing the master section and the worker section (which are located in subroutines). Only process 0, the master process, sets up this team.

```
      // In master process, run master section and worker section
      // in parallel.
      if (rank == 0)
         {
         new ParallelTeam(2).execute (new ParallelRegion()
            {
            public void run() throws Exception
```

```
               {
               execute (new ParallelSection()
                  {
                  public void run() throws Exception
                     {
                     masterSection();
                     }
                  },
               new ParallelSection()
                  {
                  public void run() throws Exception
                     {
                     workerSection();
                     }
                  });
               }
            });
         }
```

In processes 1 and higher, the one worker thread (the main program thread) merely executes the worker section.

```
      // In worker process, run only worker section.
      else
         {
         workerSection();
         }

      long t3 = System.currentTimeMillis();
```

In process 0, the parallel team threads wait at a barrier at the end of the parallel section group. When the master section has finished, meaning the master threads have sent all the tasks to the workers and have received all the results from the workers, the first team thread arrives at the barrier. When the worker section in process 0 has finished, the second team thread also arrives at the barrier. Process 0 now writes the pixel data matrix to the PJG image file and terminates; the other processes simply terminate.

```
      // Write image to PJG file in master process.
      if (rank == 0)
         {
         image = new PJGColorImage (height, width, matrix);
         PJGImage.Writer writer =
            image.prepareToWrite
               (new BufferedOutputStream
                  (new FileOutputStream (filename)));
         writer.write();
         writer.close();
         }

      // Stop timing.
      long t4 = System.currentTimeMillis();
      System.out.println (chunkCount + " chunks " + rank);
      System.out.println ((t2-t1) + " msec pre " + rank);
      System.out.println ((t3-t2) + " msec calc " + rank);
      System.out.println ((t4-t3) + " msec post " + rank);
      System.out.println ((t4-t1) + " msec total " + rank);
      }
```

Here is the subroutine with the code for the master section. Figure 23.9 shows the messages that go back and forth between the master and the worker, along with the messages' source and destination buffers.

```
/**
 * Perform the master section.
 */
private static void masterSection()
   throws IOException
   {
   int worker;
   Range range;
```

The master allocates storage for all rows of the image matrix to hold the pixel data arriving from the workers.

```
   // Allocate all rows of image matrix.
   matrix = new int [height] [width];
```

By creating a *runtime* schedule object, the master will divide the image into row slices according to the schedule specified with the `-Dpj.schedule` flag on the command line. The `start()` method call initializes the schedule object to slice up the overall row index range.

```
   // Set up a schedule object to divide the row range into
   //chunks.
   IntegerSchedule schedule = IntegerSchedule.runtime();
   schedule.start (size, new Range (0, height-1));
```

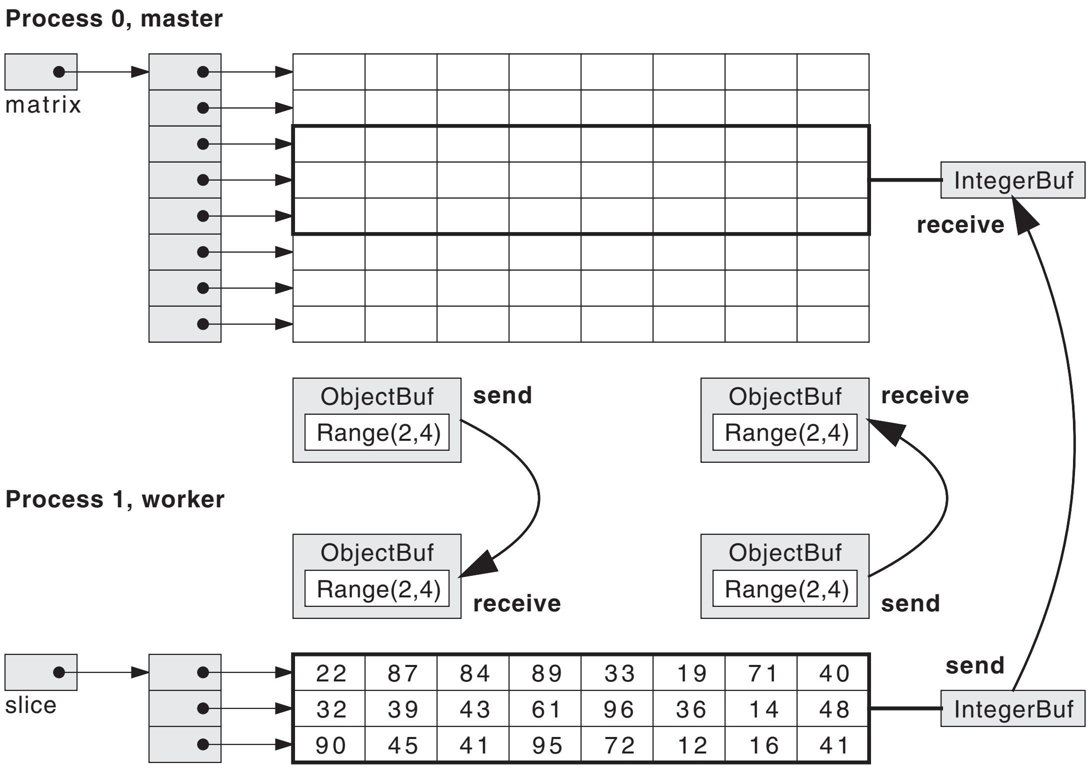


**Figure 23.9** Messages sent and received between master and worker

For each worker, the master calls the schedule object's `next()` method to get the first slice (Range object) for that worker. The master sends the Range object to the worker using a message tag of `WORKER_MSG`. If there is no slice for a worker, the `next()` method returns null, which the master duly sends to the worker to tell it there's no more work. The master keeps track of the number of active workers in the `activeWorkers` variable. Each time the master sends null to a worker, `activeWorkers` is decremented.

```
        // Send initial chunk range to each worker. If range is null,
        // no more work for that worker. Keep count of active workers.
        int activeWorkers = size;
        for (worker = 0; worker < size; ++ worker)
           {
           range = schedule.next (worker);
           world.send (worker, WORKER_MSG, ObjectBuf.buffer (range));
           if (range == null) -- activeWorkers;
           }
```

Now the master repeats its processing loop until `activeWorkers` goes to 0, indicating that there are no more slices and that all the workers have been informed of that fact.

```
      // Repeat until all workers have finished.
      while (activeWorkers > 0)
         {
```

The master receives a Range object from any worker by specifying a source rank of null in the `receive()` method call. The message tag is specified as `MASTER_MSG`; this ensures that the master receives a message with a Range object at this point rather than a message with pixel data. The master discovers which worker sent the message by examining the `fromRank` field in the communication status object returned by the `receive()` method.

```
         // Receive a chunk range from any worker.
         ObjectItemBuf<Range> rangeBuf = ObjectBuf.buffer();
         CommStatus status =
            world.receive (null, MASTER_MSG, rangeBuf);
         worker = status.fromRank;
         range = rangeBuf.item;
```

The master sets up a destination buffer encompassing the rows of the pixel data matrix specified by the range in the worker's first message. The master receives the pixel data message from the specific worker that sent the first message (not from any worker) using a message tag of `PIXEL_DATA_MSG`. The pixel data ends up in the proper rows of the matrix.

```
         // Receive pixel data from that specific worker.
         world.receive
            (worker,
             PIXEL_DATA_MSG,
             IntegerBuf.rowSliceBuffer (matrix, range));
```

Finally, the master gets the next slice for the worker from the schedule object, and sends the slice (or null if there are no more slices) to the worker. When there are no more active workers, the master exits its processing loop.

```
         // Send next chunk range to that specific worker. If null,
         // no more work.
         range = schedule.next (worker);
         world.send (worker, WORKER_MSG, ObjectBuf.buffer (range));
         if (range == null) -- activeWorkers;
         }
      }
```

Here is the subroutine with the code for the worker section.

```
   /**
    * Perform the worker section.
    */
   private static void workerSection()
      throws IOException
      {
      // Storage for matrix row slice.
      int[][] slice = null;
```

The worker repeats its processing loop until there's no more work. The worker begins by receiving a slice (Range object) from the master with a message tag of `WORKER_MSG`; if the Range object is null, the worker exits its processing loop. Out of curiosity, the worker also keeps track of the number of slices it has computed in the variable `chunkCount`; this is printed along with the running time at the end of the program.

```
      // Process chunks from master.
      for (;;)
         {
         // Receive chunk range from master. If null, no more work.
         ObjectItemBuf<Range> rangeBuf = ObjectBuf.buffer();
         world.receive (0, WORKER_MSG, rangeBuf);
         Range range = rangeBuf.item;
         if (range == null) break;
         int lb = range.lb();
         int ub = range.ub();
         int len = range.length();
         ++ chunkCount;
```

So as not to use more storage than necessary, the worker only allocates the number of rows in the slice, not the number of rows in the whole image. Further, to save time, the worker reuses the previous slice's storage if it has enough rows to hold the current slice.

```
         // Allocate storage for matrix row slice if necessary.
         if (slice == null || slice.length < len)
            {
            slice = new int [len] [width];
            }
```

The worker proceeds to compute each pixel in rows `lb` through `ub`. However, these are the row indexes in the full pixel matrix back in the master. The row indexes in the worker's slice go from 0 to `len-1`, not from `lb` to `ub`. Thus, the reference to the current row (`slice_r`) is set to `slice[r-lb]`, not `slice[r]`.

```
            // Compute all rows and columns in slice.
            for (int r = lb; r <= ub; ++ r)
               {
               int[] slice_r = slice[r-lb];
               double y = ycenter + (yoffset - r) / resolution;

               for (int c = 0; c < width; ++ c)
                  {
                  double x = xcenter + (xoffset + c) / resolution;

                  // Iterate until convergence.
                  int i = 0;
                  double aold = 0.0;
                  double bold = 0.0;
                  double a = 0.0;
                  double b = 0.0;
                  double zmagsqr = 0.0;
                  while (i < maxiter && zmagsqr <= 4.0)
                     {
                     ++ i;
                     a = aold*aold - bold*bold + x;
                     b = 2.0*aold*bold + y;
                     zmagsqr = a*a + b*b;
                     aold = a;
                     bold = b;
                     }

                  // Record number of iterations for pixel.
                  slice_r[c] = huetable[i];
                  }
               }
```

Here is the first of the two messages the worker sends to report its result to the master. The Range object that came from the master is still sitting in the destination buffer, `rangeBuf`. The worker simply uses `rangeBuf` as the source buffer to send the Range object back to the master with a message tag of `MASTER_MSG`.

```
            // Send chunk range back to master.
            world.send (0, MASTER_MSG, rangeBuf);
```

The worker sets up a source buffer encompassing row indexes 0 through `len-1` of the slice matrix, and sends the pixel data to the master with a message tag of `PIXEL_DATA_MSG`. The worker then repeats its processing loop.

```
            // Send pixel data to master.
            world.send
               (0,
                PIXEL_DATA_MSG,
                IntegerBuf.rowSliceBuffer
                   (slice, new Range (0, len-1)));
            }
         };
      }
```

This concludes the parallel master-worker Mandelbrot Set program.

Table 23.2 (at the end of the chapter) lists, and Figure 23.10 plots, the MandelbrotSetClu2 program's running-time data on the "tardis" cluster. Each program run used the same command-line arguments as Table 23.1 and Figure 23.3, with a dynamic schedule for load balancing.

```
    $ java -Dpj.schedule="dynamic(10)" . . .
```

The data show that the master-worker pattern with a dynamic schedule has balanced the load. The running times steadily decrease as the number of processors increases. However, the speedups flatten out rather quickly, symptomatic of a substantial sequential fraction. Where is the sequential fraction coming from?

Several factors contribute to the sequential fraction, including the initialization that takes place before the master and worker processing commences, the allocation of the pixel data matrix or slice at the beginning of the master and worker processing, and the generation of the PJG image file after the master and worker processing finishes. But in a cluster parallel program, there is an additional contribution to the sequential fraction not found in an SMP parallel program: the time spent in message passing. While the worker is sending its slice message and its pixel data message to the master and is waiting to receive its next slice message from the master, the worker is not computing any pixels. This increases the parallel version's running time relative to the sequential version, thus reducing the speedups, reducing the efficiencies, and increasing the *EDSFs*. In Chapter 24, we will quantify just how much time this program spends in message passing.

Before leaving the running-time metrics, observe the strangely high efficiency, about 1.1, of the parallel version running in one processor. This is another manifestation of the JIT compiler effect. Recall that there are two threads in process 0, the master thread and the worker thread; the multiple threads let the JIT compiler detect and compile hot spots more quickly. But when running in two or more processors, the processes at rank 1 and higher have only one thread, the JIT compiler does not optimize these processes as quickly as process 0, and the efficiencies drop back below 1.0. Because the running time on one processor is smaller than it would be without the JIT compiler effect, the *EDSF* values for smaller numbers of processors are higher than they would be otherwise. As more processors are added, the *EDSF* curves converge to roughly horizontal lines, reflecting a constant sequential fraction.

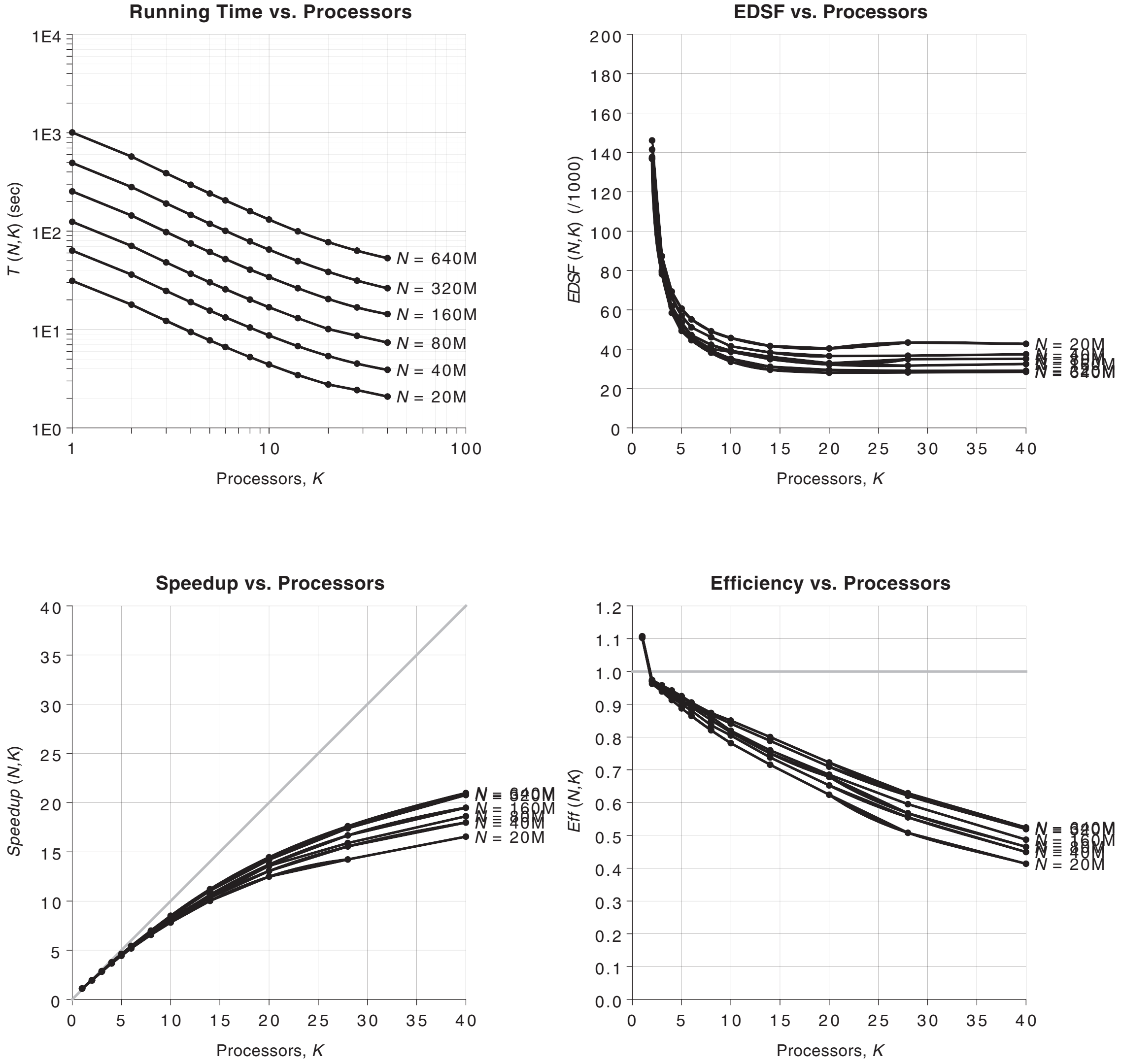


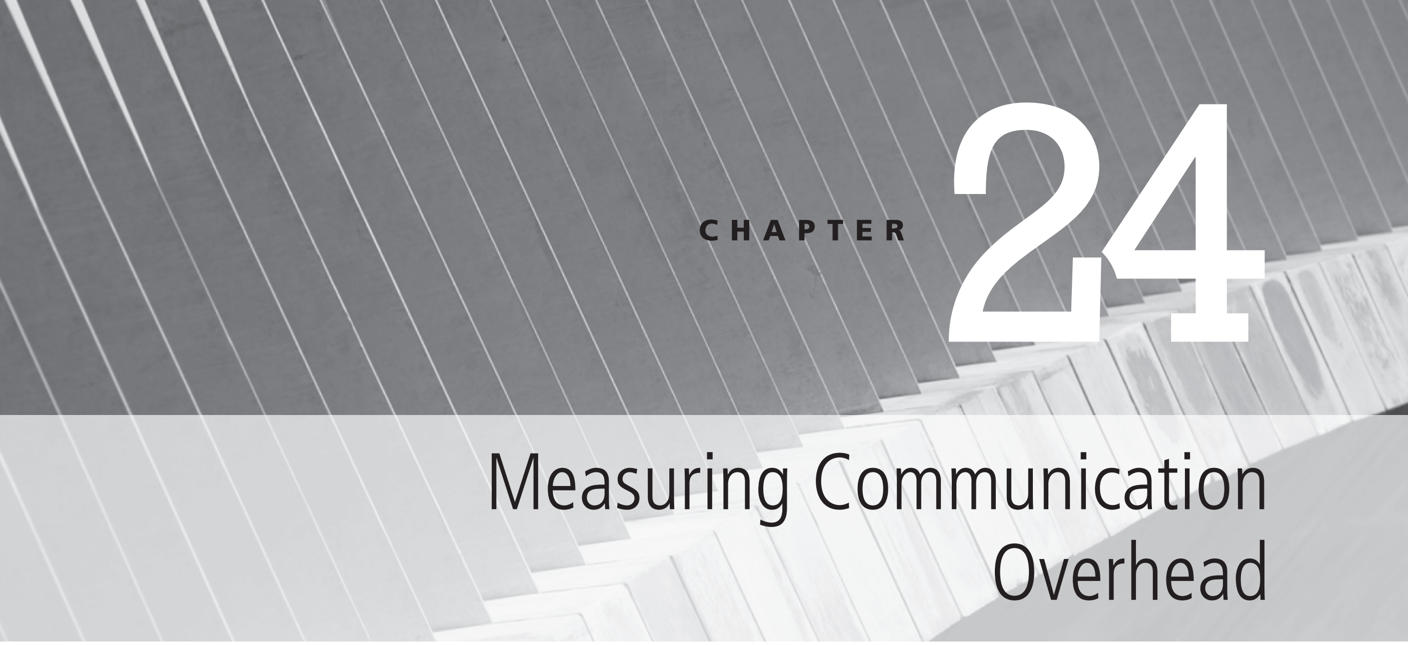
**Figure 23.10** MandelbrotSetSeq/MandelbrotSetClu2 running-time metrics

**Table 23.1** MandelbrotSetSeq/MandelbrotSetClu running-time metrics

| *N* | *K* | *T* | *Spdup* | *Eff* | *EDSF* | *N* | *K* | *T* | *Spdup* | *Eff* | *EDSF* |
|---:|---:|---:|---:|---:|---:|---:|---:|---:|---:|---:|---:|
| 20M | seq | 34463 | | | | 160M | seq | 279672 | | | |
| 20M | 1 | 34474 | 1.000 | 1.000 | | 160M | 1 | 280467 | 0.997 | 0.997 | |
| 20M | 2 | 17648 | 1.953 | 0.976 | 0.024 | 160M | 2 | 142849 | 1.958 | 0.979 | 0.019 |
| 20M | 3 | 26136 | 1.319 | 0.440 | 0.637 | 160M | 3 | 212524 | 1.316 | 0.439 | 0.637 |
| 20M | 4 | 16635 | 2.072 | 0.518 | 0.310 | 160M | 4 | 134843 | 2.074 | 0.519 | 0.308 |
| 20M | 5 | 18166 | 1.897 | 0.379 | 0.409 | 160M | 5 | 146996 | 1.903 | 0.381 | 0.405 |
| 20M | 6 | 13511 | 2.551 | 0.425 | 0.270 | 160M | 6 | 109151 | 2.562 | 0.427 | 0.267 |
| 20M | 8 | 11177 | 3.083 | 0.385 | 0.228 | 160M | 8 | 90012 | 3.107 | 0.388 | 0.224 |
| 20M | 10 | 9469 | 3.640 | 0.364 | 0.194 | 160M | 10 | 77005 | 3.632 | 0.363 | 0.194 |
| 20M | 14 | 7191 | 4.793 | 0.342 | 0.148 | 160M | 14 | 58418 | 4.787 | 0.342 | 0.147 |
| 20M | 20 | 5189 | 6.642 | 0.332 | 0.106 | 160M | 20 | 42870 | 6.524 | 0.326 | 0.108 |
| 20M | 28 | 4025 | 8.562 | 0.306 | 0.084 | 160M | 28 | 32127 | 8.705 | 0.311 | 0.082 |
| 20M | 40 | 3088 | 11.160 | 0.279 | 0.066 | 160M | 40 | 24084 | 11.612 | 0.290 | 0.062 |
| 40M | seq | 70096 | | | | 320M | seq | 547023 | | | |
| 40M | 1 | 70063 | 1.000 | 1.000 | | 320M | 1 | 548300 | 0.998 | 0.998 | |
| 40M | 2 | 35850 | 1.955 | 0.978 | 0.023 | 320M | 2 | 278781 | 1.962 | 0.981 | 0.017 |
| 40M | 3 | 53276 | 1.316 | 0.439 | 0.641 | 320M | 3 | 415362 | 1.317 | 0.439 | 0.636 |
| 40M | 4 | 33831 | 2.072 | 0.518 | 0.310 | 320M | 4 | 263352 | 2.077 | 0.519 | 0.307 |
| 40M | 5 | 36967 | 1.896 | 0.379 | 0.410 | 320M | 5 | 288072 | 1.899 | 0.380 | 0.407 |
| 40M | 6 | 27383 | 2.560 | 0.427 | 0.269 | 320M | 6 | 213029 | 2.568 | 0.428 | 0.266 |
| 40M | 8 | 22630 | 3.097 | 0.387 | 0.226 | 320M | 8 | 175619 | 3.115 | 0.389 | 0.223 |
| 40M | 10 | 19443 | 3.605 | 0.361 | 0.197 | 320M | 10 | 150382 | 3.638 | 0.364 | 0.194 |
| 40M | 14 | 14651 | 4.784 | 0.342 | 0.148 | 320M | 14 | 114177 | 4.791 | 0.342 | 0.147 |
| 40M | 20 | 10729 | 6.533 | 0.327 | 0.109 | 320M | 20 | 83587 | 6.544 | 0.327 | 0.108 |
| 40M | 28 | 8148 | 8.603 | 0.307 | 0.084 | 320M | 28 | 62723 | 8.721 | 0.311 | 0.082 |
| 40M | 40 | 6159 | 11.381 | 0.285 | 0.065 | 320M | 40 | 46786 | 11.692 | 0.292 | 0.062 |
| 80M | seq | 137327 | | | | 640M | seq | 1116057 | | | |
| 80M | 1 | 137311 | 1.000 | 1.000 | | 640M | 1 | 1119359 | 0.997 | 0.997 | |
| 80M | 2 | 70151 | 1.958 | 0.979 | 0.022 | 640M | 2 | 571346 | 1.953 | 0.977 | 0.021 |
| 80M | 3 | 104254 | 1.317 | 0.439 | 0.639 | 640M | 3 | 852306 | 1.309 | 0.436 | 0.642 |
| 80M | 4 | 66142 | 2.076 | 0.519 | 0.309 | 640M | 4 | 542763 | 2.056 | 0.514 | 0.313 |
| 80M | 5 | 72216 | 1.902 | 0.380 | 0.407 | 640M | 5 | 593785 | 1.880 | 0.376 | 0.413 |
| 80M | 6 | 53443 | 2.570 | 0.428 | 0.267 | 640M | 6 | 439437 | 2.540 | 0.423 | 0.271 |
| 80M | 8 | 44175 | 3.109 | 0.389 | 0.225 | 640M | 8 | 358484 | 3.113 | 0.389 | 0.223 |
| 80M | 10 | 37819 | 3.631 | 0.363 | 0.195 | 640M | 10 | 306056 | 3.647 | 0.365 | 0.193 |
| 80M | 14 | 28772 | 4.773 | 0.341 | 0.149 | 640M | 14 | 233446 | 4.781 | 0.341 | 0.148 |
| 80M | 20 | 20966 | 6.550 | 0.327 | 0.108 | 640M | 20 | 170445 | 6.548 | 0.327 | 0.108 |
| 80M | 28 | 15772 | 8.707 | 0.311 | 0.082 | 640M | 28 | 127722 | 8.738 | 0.312 | 0.081 |
| 80M | 40 | 11850 | 11.589 | 0.290 | 0.063 | 640M | 40 | 95390 | 11.700 | 0.292 | 0.062 |

**Table 23.2** MandelbrotSetSeq/MandelbrotSetClu2 running-time metrics

| *N* | *K* | *T* | *Spdup* | *Eff* | *EDSF* | *N* | *K* | *T* | *Spdup* | *Eff* | *EDSF* |
|---:|---:|---:|---:|---:|---:|---:|---:|---:|---:|---:|---:|
| 20M | seq | 34464 | | | | 160M | seq | 279855 | | | |
| 20M | 1 | 31247 | 1.103 | 1.103 | | 160M | 1 | 253410 | 1.104 | 1.104 | |
| 20M | 2 | 17906 | 1.925 | 0.962 | 0.146 | 160M | 2 | 144145 | 1.941 | 0.971 | 0.138 |
| 20M | 3 | 12233 | 2.817 | 0.939 | 0.087 | 160M | 3 | 97811 | 2.861 | 0.954 | 0.079 |
| 20M | 4 | 9436 | 3.652 | 0.913 | 0.069 | 160M | 4 | 75088 | 3.727 | 0.932 | 0.062 |
| 20M | 5 | 7766 | 4.438 | 0.888 | 0.061 | 160M | 5 | 61522 | 4.549 | 0.910 | 0.053 |
| 20M | 6 | 6644 | 5.187 | 0.865 | 0.055 | 160M | 6 | 51951 | 5.387 | 0.898 | 0.046 |
| 20M | 8 | 5249 | 6.566 | 0.821 | 0.049 | 160M | 8 | 40578 | 6.897 | 0.862 | 0.040 |
| 20M | 10 | 4410 | 7.815 | 0.781 | 0.046 | 160M | 10 | 34161 | 8.192 | 0.819 | 0.039 |
| 20M | 14 | 3440 | 10.019 | 0.716 | 0.042 | 160M | 14 | 26318 | 10.634 | 0.760 | 0.035 |
| 20M | 20 | 2761 | 12.482 | 0.624 | 0.040 | 160M | 20 | 20420 | 13.705 | 0.685 | 0.032 |
| 20M | 28 | 2423 | 14.224 | 0.508 | 0.043 | 160M | 28 | 16786 | 16.672 | 0.595 | 0.032 |
| 20M | 40 | 2083 | 16.545 | 0.414 | 0.043 | 160M | 40 | 14363 | 19.484 | 0.487 | 0.032 |
| 40M | seq | 70084 | | | | 320M | seq | 546761 | | | |
| 40M | 1 | 63395 | 1.106 | 1.106 | | 320M | 1 | 493854 | 1.107 | 1.107 | |
| 40M | 2 | 36182 | 1.937 | 0.968 | 0.141 | 320M | 2 | 280704 | 1.948 | 0.974 | 0.137 |
| 40M | 3 | 24679 | 2.840 | 0.947 | 0.084 | 320M | 3 | 190801 | 2.866 | 0.955 | 0.080 |
| 40M | 4 | 19011 | 3.686 | 0.922 | 0.067 | 320M | 4 | 146233 | 3.739 | 0.935 | 0.061 |
| 40M | 5 | 15573 | 4.500 | 0.900 | 0.057 | 320M | 5 | 118798 | 4.602 | 0.920 | 0.051 |
| 40M | 6 | 13266 | 5.283 | 0.880 | 0.051 | 320M | 6 | 100744 | 5.427 | 0.905 | 0.045 |
| 40M | 8 | 10481 | 6.687 | 0.836 | 0.046 | 320M | 8 | 78733 | 6.944 | 0.868 | 0.039 |
| 40M | 10 | 8707 | 8.049 | 0.805 | 0.041 | 320M | 10 | 65015 | 8.410 | 0.841 | 0.035 |
| 40M | 14 | 6785 | 10.329 | 0.738 | 0.038 | 320M | 14 | 49531 | 11.039 | 0.788 | 0.031 |
| 40M | 20 | 5370 | 13.051 | 0.653 | 0.037 | 320M | 20 | 38521 | 14.194 | 0.710 | 0.029 |
| 40M | 28 | 4507 | 15.550 | 0.555 | 0.037 | 320M | 28 | 31435 | 17.393 | 0.621 | 0.029 |
| 40M | 40 | 3898 | 17.979 | 0.449 | 0.037 | 320M | 40 | 26315 | 20.778 | 0.519 | 0.029 |
| 80M | seq | 137256 | | | | 640M | seq | 1115607 | | | |
| 80M | 1 | 124461 | 1.103 | 1.103 | | 640M | 1 | 1007507 | 1.107 | 1.107 | |
| 80M | 2 | 70767 | 1.940 | 0.970 | 0.137 | 640M | 2 | 572637 | 1.948 | 0.974 | 0.137 |
| 80M | 3 | 48130 | 2.852 | 0.951 | 0.080 | 640M | 3 | 388308 | 2.873 | 0.958 | 0.078 |
| 80M | 4 | 36891 | 3.721 | 0.930 | 0.062 | 640M | 4 | 296031 | 3.769 | 0.942 | 0.058 |
| 80M | 5 | 30228 | 4.541 | 0.908 | 0.054 | 640M | 5 | 241254 | 4.624 | 0.925 | 0.049 |
| 80M | 6 | 25635 | 5.354 | 0.892 | 0.047 | 640M | 6 | 205340 | 5.433 | 0.905 | 0.045 |
| 80M | 8 | 20161 | 6.808 | 0.851 | 0.042 | 640M | 8 | 159642 | 6.988 | 0.874 | 0.038 |
| 80M | 10 | 16835 | 8.153 | 0.815 | 0.039 | 640M | 10 | 131208 | 8.503 | 0.850 | 0.034 |
| 80M | 14 | 13069 | 10.502 | 0.750 | 0.036 | 640M | 14 | 99607 | 11.200 | 0.800 | 0.030 |
| 80M | 20 | 10109 | 13.578 | 0.679 | 0.033 | 640M | 20 | 77222 | 14.447 | 0.722 | 0.028 |
| 80M | 28 | 8636 | 15.893 | 0.568 | 0.035 | 640M | 28 | 63372 | 17.604 | 0.629 | 0.028 |
| 80M | 40 | 7374 | 18.614 | 0.465 | 0.035 | 640M | 40 | 53195 | 20.972 | 0.524 | 0.029 |



# CHAPTER 24

# Measuring Communication Overhead

in which we measure how long it takes to send a message from one processor to another; we derive a mathematical model for communication time; and we consider the implications for cluster parallel program design

# CHAPTER 24 Measuring Communication Overhead

## 24.1 Measuring the Time to Send a Message

To determine how much time the MandelbrotSetClu2 program—or any cluster parallel program—spends doing message passing, we need a formula that can predict how long it takes to send all the messages in the program, given the number of messages and the number of data elements in each message. To derive such a formula, we first must measure the time required to send a message. Is there a way to do that without resorting to cumbersome low-level network analyzers or packet sniffers?

A typical technique uses **ping-pong messages**. The ping-pong program runs in two processes on the cluster parallel computer being measured. Figure 24.1 is a space-time diagram showing the two processes across the top (the space dimension) and showing time *t* increasing down the page. Process 0 sends a message of a certain size *n* to process 1. As soon as it finishes receiving the message, process 1 sends the same message back to process 0. Process 0 measures how much time elapses from when it started sending the message until it finished receiving the reply; this is the message's **round-trip time** *RTT*. The message's **one-way time** *T* is then *RTT*/2; this is the quantity we are interested in. By measuring *T* for messages of various sizes *n*, we can map out *T* as a function of *n*, namely *T*(*n*).

Because *T* might be just a few microseconds, but the system clock's resolution is typically on the order of milliseconds, it's difficult to measure *T* for just one message. Instead, the ping-pong program sends and receives the message repeatedly and measures the time to do all the repetitions. If the total of all the round-trip times is *TRTT* and the number of repetitions is *R*, then a single round trip takes *RTT* = *TRTT*/*R*, and the message one-way time is *T* = *TRTT*/2*R*. By doing, say, *R* = 10,000 repetitions, we can get a reasonably accurate measurement for *T*.

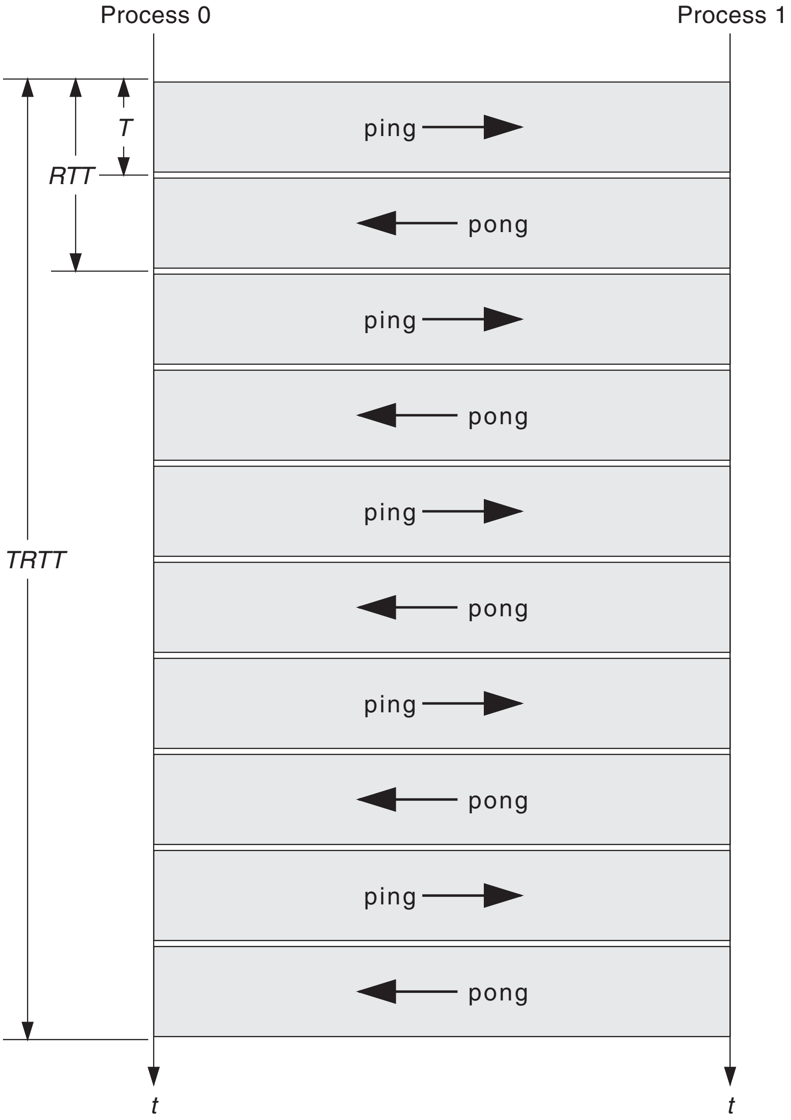


**Figure 24.1** Measuring message time with ping-pong messages

Class edu.rit.clu.timing.TimeSendByte is a ping-pong program that sends messages with data items of type byte. The command-line arguments are the number of repetitions *R* followed by one or more values for the message size *n*, where *n* is the number of byte data items in the message. Here is the source code.

```
package edu.rit.clu.timing;
import edu.rit.mp.ByteBuf;
import edu.rit.pj.Comm;
import java.io.IOException;
import java.text.DecimalFormat;
import java.util.Date;
public class TimeSendByte
   {
   private static final DecimalFormat FMT3 =
      new DecimalFormat ("0.00E0");

   static Comm world;
   static int size;
   static int rank;

   static int reps;
   static int numn;
   static int[] n;

   /**
    * Main program.
    */
   public static void main
      (String[] args)
      throws Exception
      {
      // Initialize PJ.
      Comm.init (args);
      world = Comm.world();
      size = world.size();
      rank = world.rank();

      // Parse command line arguments.
      if (args.length < 2) usage();
      reps = Integer.parseInt (args[0]);
      numn = args.length - 1;
      n = new int [numn];
      for (int i = 0; i < numn; ++ i)
         {
         n[i] = Integer.parseInt (args[i+1]);
         }
```

Process 0 sends, and then receives messages and measures the one-way time *T* for each value of *n*.

```
      // Process 0.
      if (rank == 0)
         {
         System.out.println
         ("java -Dpj.np=2 edu.rit.clu.timing.TimeSendByte " +
          reps);
      System.out.println (new Date());
      System.out.println ("n\ttime1\ttime2\tSend time (sec)");

      // Test each value of n.
      for (int i = 0; i < numn; ++ i)
         {
         // Create message buffer.
         int n_i = n[i];
         byte[] bufarray = new byte [n_i];
         ByteBuf buf = ByteBuf.buffer (bufarray);
```

Each message contains data values of 0, 1, 2, and so on. Here, we measure the time it takes to do *R* repetitions of just filling in the data source array, without sending or receiving any messages. (The code to do this is in a subroutine.) Later, we subtract this overhead time (`time1`) from the running-time measurement.

```
         // Time repetitions without sending messages.
         long time1 = -System.currentTimeMillis();
         for (int j = 0; j < reps; ++ j)
            {
            fill (n_i, bufarray, buf);
            }
         time1 += System.currentTimeMillis();
```

And here we measure the time it takes to do *R* repetitions of filling in the data source array, sending the message to process 1, and receiving the message back from process 1 (`time2`).

```
         // Time repetitions with sending messages.
         long time2 = -System.currentTimeMillis();
         for (int j = 0; j < reps; ++ j)
            {
            fillSendReceive (n_i, bufarray, buf);
            }
         time2 += System.currentTimeMillis();
```

The message one-way time *T* is *TRTT*/2*R*, where *TRTT* is the difference between `time2` and `time1`. The result is divided by a further factor of 1,000 to convert *T* from units of milliseconds to units of seconds.

```
         // Print results.
         double sendtime =
            ((double)(time2-time1)) / ((double) reps) / 2000.0;
         System.out.print (n_i);
```

```
            System.out.print ('\t');
            System.out.print (time1);
            System.out.print ('\t');
            System.out.print (time2);
            System.out.print ('\t');
            System.out.print (FMT3.format (sendtime));
            System.out.println();
            }
         }
```

Here is the code executed in process 1. It is almost identical to the code for process 0, except process 1 first receives each message, and then sends it back.

```
      // Process 1.
      else if (rank == 1)
         {
         // Test each value of n.
         for (int i = 0; i < numn; ++ i)
            {
            // Create message buffer.
            int n_i = n[i];
            byte[] bufarray = new byte [n_i];
            ByteBuf buf = ByteBuf.buffer (bufarray);

            // Do repetitions without receiving messages.
            for (int j = 0; j < reps; ++ j)
               {
               fill (n_i, bufarray, buf);
               }

            // Do repetitions with receiving messages.
            for (int j = 0; j < reps; ++ j)
               {
               fillReceiveSend (n_i, bufarray, buf);
               }
            }
         }
      }
```

Finally, here are the three subroutines the prior code calls.

```
   /**
    * Fill the buffer.
    */
```

```
private static void fill
   (int n_i,
    byte[] bufarray,
    ByteBuf buf)
   {
   for (int k = 0; k < n_i; ++ k)
      {
      bufarray[k] = (byte) k;
      }
   }

/**
 * Fill the buffer, send a message, receive a message.
 */
private static void fillSendReceive
   (int n_i,
    byte[] bufarray,
    ByteBuf buf)
   throws IOException
   {
   for (int k = 0; k < n_i; ++ k)
      {
      bufarray[k] = (byte) k;
      }
   world.send (1, buf);
   world.receive (1, buf);
   }

/**
 * Fill the buffer, receive a message, send a message.
 */
private static void fillReceiveSend
   (int n_i,
    byte[] bufarray,
    ByteBuf buf)
   throws IOException
   {
   for (int k = 0; k < n_i; ++ k)
      {
      bufarray[k] = (byte) k;
      }
   world.receive (0, buf);
   world.send (0, buf);
   }
}
```

The Parallel Java Library also includes the classes TimeSendInt and TimeSendDouble, which are identical to class TimeSendByte, except that they send messages consisting of type `int` and `double`, respectively.

## 24.2 Message Send-Time Model

Here is what the TimeSendByte program printed for one run on two processors of the "tardis" hybrid parallel computer:

```
$ java -Dpj.np=2 edu.rit.clu.timing.TimeSendByte 10000 \
  1000 1000 1000 100 100 100 200 200 200 500 500 500 1000 1000 1000 \
  2000 2000 2000 5000 5000 5000 10000 10000 10000 20000 20000 20000 \
  50000 50000 50000 100000 100000 100000 200000 200000 200000 \
  500000 500000 500000 1000000 1000000 1000000
Job 1, dr01, dr02
java -Dpj.np=2 edu.rit.clu.timing.TimeSendByte 10000
Wed Dec 26 14:59:42 EST 2007
n       time1   time2   Send time (sec)
1000    65      3169    1.55E-4
1000    8       3042    1.52E-4
1000    8       3036    1.51E-4
100     0       2295    1.15E-4
100     1       2673    1.34E-4
100     0       2673    1.34E-4
. . .
1000000 8700    185052  8.82E-3
1000000 8695    185046  8.82E-3
1000000 8699    185035  8.82E-3
```

(The first three *n* values are included to "warm up" the JVM. Notice how the measured times decrease as the JIT compiler works its magic. We will disregard these first three measurements.) Table 24.1 (at the end of the chapter) gives the complete list of measurements, along with measurements from the TimeSendInt and TimeSendDouble programs.

Figure 24.2 plots the data in Table 24.1. The horizontal axis is the message size *B* in units of *bits.* For the TimeSendByte program, *B* ranged from 800 to 8,000,000 bits; for TimeSendInt, 3,200 to 32,000,000 bits; for TimeSendDouble, 6,400 to 64,000,000 bits. The vertical axis is the message send time *T* in units of seconds. The two plots in Figure 24.2 show the same data on two different scales. The first plot is for $B \le 800{,}000$ bits; the second plot is for $B \le 64{,}000{,}000$ bits.

Figure 24.2 also plots the linear regression of the data. The regression formula is the following:

$$T(B) = 2.08 \times 10^{-4} + 1.07 \times 10^{-9} B \qquad (24.1)$$

Equation 24.1 is the **message send-time model**—a formula for $T$ (in seconds) as a function of $B$ (in bits)—for the “tardis” computer. The correlation coefficient is 0.999973, evincing good agreement between the model and the data. The constant term in Equation 24.1 is apparent in the first plot in Figure 24.2 as the nonzero intercept of the regression line; because of the vertical scale, the nonzero intercept is difficult to see in the second plot.

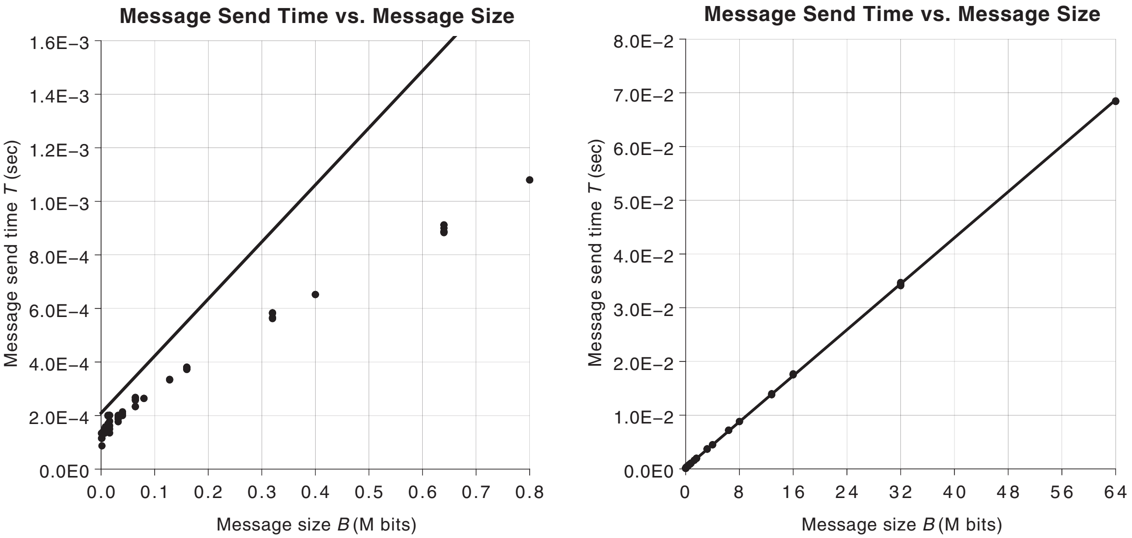


**Figure 24.2** Message send-time data and model—smaller message sizes, larger message sizes

Let’s consider the two terms in the message send-time model. The constant term, $2.08\times10^{-4}$ sec (208 μsec), represents the **message latency**. This includes items such as the time needed to execute the code in the Parallel Java communication layer to initiate sending the message, and the backend network’s latency. The message latency is incurred on every message sent, no matter how large or small the message.

The $O(B)$ term, $1.07\times10^{-9}\,B$ sec, represents the **message transmission time**. Once past the fixed message latency, it takes $1.07\times10^{-9}$ sec to send each bit of message data. The **message bandwidth** in bits per second is the reciprocal of the coefficient of $B$: 0.937 Gbps. But the “tardis” computer’s backend network bandwidth is 1 Gbps. Why is the message bandwidth less than the network bandwidth?

The diminished bandwidth is partly due to the **protocol overhead** from the Ethernet, IP, TCP, and application-layer protocols Parallel Java uses to send messages. Figure 24.3 shows the bytes that actually go out over the wire when a message is transmitted. First comes the inter-frame interval, during which time (12 bytes’ worth) the Ethernet interface does not transmit anything; this is to let the communication medium recover from the previous transmission. Then comes the Ethernet preamble (8 bytes), used to synchronize the Ethernet transmitter’s and receiver’s clocks. After that come the Ethernet header (14 bytes), IP header (20 bytes), TCP header (20 bytes), and Parallel Java message header (13 bytes). At this point, finally, the message data is transmitted. The maximum amount of information allowed in an Ethernet frame—the **maximum transmission unit (MTU)**—is 1,500 bytes. This includes the IP header, TCP header, and Parallel Java message (header plus data). If the message is too long to fit into a single Ethernet frame, it is the TCP layer's job to break the message into **segments** and transmit each segment in a separate Ethernet frame. The maximum amount of data in a TCP segment—the **maximum segment size (MSS)**—is 1,460 bytes, namely the MTU minus the IP and TCP headers. Thus, the largest message that can fit in an Ethernet frame is 1,460 bytes long—13 bytes of header plus 1,447 bytes of data. After the message comes the Ethernet frame check sequence (4 bytes) used to detect transmission errors, and that concludes the frame.

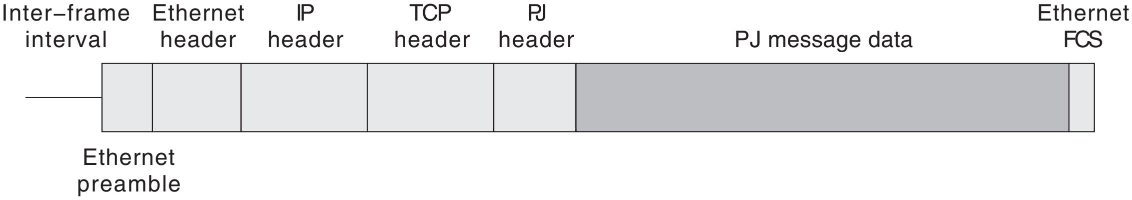


**Figure 24.3** Protocol overhead in a Parallel Java message

To send an MSS's worth of message data necessitates 91 bytes of protocol overhead. For a single-frame message, the effective message bandwidth is 1,447/(1,447+91) times the raw network bandwidth—0.941 Gbps. The protocol overhead alone causes a 5.9 percent reduction in the rate at which we can send data.

Other factors besides the protocol overhead reduce the effective message bandwidth. These include: the time needed for the sending process to copy the message data from the message's source buffer to the outgoing TCP buffer; any time the sending TCP layer has to wait to receive an acknowledgment for a TCP segment from the receiving TCP layer; any time the sending TCP layer has to stop sending data temporarily because the receiving TCP layer's incoming buffer is full (**flow control**); and the time needed for the receiving process to copy the message data from the incoming TCP buffer to the message's destination buffer. These factors increase the amount of time needed to send one byte, thus decreasing the effective message bandwidth still further. We measured an overall message bandwidth reduction of 6.3 percent—0.937 Gbps instead of 1 Gbps.

From the message send-time model, Equation 24.1, we can gain two important insights regarding cluster parallel program design. First, to get good parallel performance, *the program must have much more computation than communication.* With today's CPU clock speeds, a program can do a *lot* of computation in the time it takes to send a single message, even on a high-bandwidth cluster backend network. Unless the program does a lot of computation between each message, a large portion of the program's running time will be occupied with sending messages; this communication time—not present in the sequential version—will decrease the parallel version's speedups and efficiencies.

It's best if the program's asymptotic communication time as a function of the problem size parameter $n$ is of a smaller order than the program's asymptotic computation time. If, for example, the computation time is $O(n^3)$ while the communication time is only $O(n^2)$, as the problem size scales up, the computation time eventually dominates the communication time, the sequential fraction due to communication is small, and the parallel performance is good. This is another manifestation of the *surface-to-volume effect* we saw in Chapter 16 ("surface" being the communication and "volume" the computation).

On the other hand, if the program's asymptotic communication time is of the same order as the asymptotic computation time, there will be no surface-to-volume effect. In this case the amount of time spent computing must be much larger than the amount of time spent communicating to get a small sequential fraction and good parallel performance.

The second insight regarding cluster parallel program design is that *a few large messages are better than many small messages.* This is because the message latency, the constant coefficient in the message send-time model, is typically several orders of magnitude larger than the linear term's coefficient. If the messages are small, most of the message send time is spent on the message latency rather than on useful data transmission; the more messages there are, the more times the message latency is incurred. If possible, it is better to store data temporarily and send the data all at once in a single large message than to send the data piecemeal in several small messages. The trade-off, however, is that storing data temporarily might increase the program's memory requirements.

## 24.3 Applying the Model

Now that we have a message send-time model for the "tardis" computer, let's use the model to analyze the MandelbrotSetClu2 program's running time measurements from Chapter 23. The sequential version's running time to compute a 40M pixel image was the following:

```
$ java edu.rit.clu.fractal.MandelbrotSetSeq 6400 6400 -0.75 0.0 2400 \
  1000 0.4 image.pjg
655 msec pre
68992 msec calc
437 msec post
70084 msec total
```

The parallel version's running time on two processors, using a dynamic schedule with a chunk size of 10, was the following:

```
$ java -Dpj.np=2 -Dpj.schedule="dynamic(10)" \
  edu.rit.clu.fractal.MandelbrotSetClu2 6400 6400 -0.75 0.0 2400 \
  1000 0.4 image.pjg
287 chunks 1
10 msec pre 1
35726 msec calc 1
0 msec post 1
35736 msec total 1
353 chunks 0
11 msec pre 0
35721 msec calc 0
450 msec post 0
36182 msec total 0
```

Whereas chunks computed by process 0 require no message passing over the network between the master thread and the worker thread, chunks computed by process 1 do require message passing over the network. For each chunk, the master thread in process 0 sends a message containing one Range object to the worker thread in process 1; the worker sends a message containing one Range object back to the master; and the worker sends a message containing 64,000 integers (10 rows times 6,400 columns of pixel values) to the master thread. According to the preceding printout, process 1 computed 287 chunks.

To estimate how much time the program spent doing message passing, we use the message send-time model. A serialized Range object occupies 54 bytes, or $B = 432$ bits. A message with 64,000 integers contains $B = 2{,}048{,}000$ bits. Thus, the model predicts that the time to send all the messages was the following:

$$\begin{aligned}&287\cdot(2.08\times10^{-4} + 1.07\times10^{-9}\cdot432 +\\&\qquad 2.08\times10^{-4} + 1.07\times10^{-9}\cdot432 +\\&\qquad 2.08\times10^{-4} + 1.07\times10^{-9}\cdot2{,}048{,}000)\\&= 808 \text{ msec}\end{aligned}$$

On one processor, the MandelbrotSetSeq program took 655+437 = 1,092 msec for the sequential portion and 68,992 msec for the parallelizable portion. On two processors, assume that doing 353 chunks in process 0 and 287 chunks in process 1 resulted in a balanced load. Then, on two processors, the MandelbrotSetClu2 program should have taken the same amount of time for the sequential portion, 1,092 msec, plus half as much time for the parallelizable portion, 68,992÷2 = 34,496 msec. However, the latter time was increased by the message-passing time of 808 msec. Thus, the whole thing should have taken 1,092+34,496+808 = 36,396 msec. This estimate is within 0.6 percent of the measured value, 36,182 msec.

Let’s consider how the message passing affected the program’s sequential fraction. From the sequential version’s running-time measurements, we would have expected a sequential fraction of (655+437)/(655+437+68,992) = 0.016. But because of the message passing, the sequential fraction actually was (655+437+808)/(655+437+808+68,992) = 0.027—nearly twice as much! Without message passing, we would have expected a maximum speedup of 1/0.016 = 64; with message passing, the actual maximum speedup was 1/0.027 = 37.

The master-worker pattern improves the Mandelbrot Set program’s performance by balancing the load. However, the additional message passing reduces the Mandelbrot Set program’s performance by increasing the sequential fraction. Might there be a way to achieve a balanced load without incurring such a high performance penalty?

## 24.4 Design with Reduced Message Passing

In the MandelbrotSetClu2 program, the messages conveying pixel data from the workers to the master occupy most of the message-passing time. If we want a significant reduction in the message passing, we have to get rid of these messages. But these messages are needed to collect all the pixel data in one matrix, so the master process can write the PJG image file. What can we do differently?

One alternative is for *each worker to write a separate image file,* instead of having the master write a single image file. That is, each worker writes the row slices it computes directly to its own image file, in parallel with all the other workers writing their image files. With this design, there’s no longer any need for the workers to send pixel data messages to the master. To balance the load, the master must still send row slices (Range objects) to the workers, and each worker must still notify the master when the worker has finished computing a slice. (The latter messages need not carry any data.) Figure 24.4 shows this design's overall execution timeline. It is similar to Figure 23.8, except that after notifying the master, a worker writes the just-computed slice to its image file while waiting for the master to send the next slice.

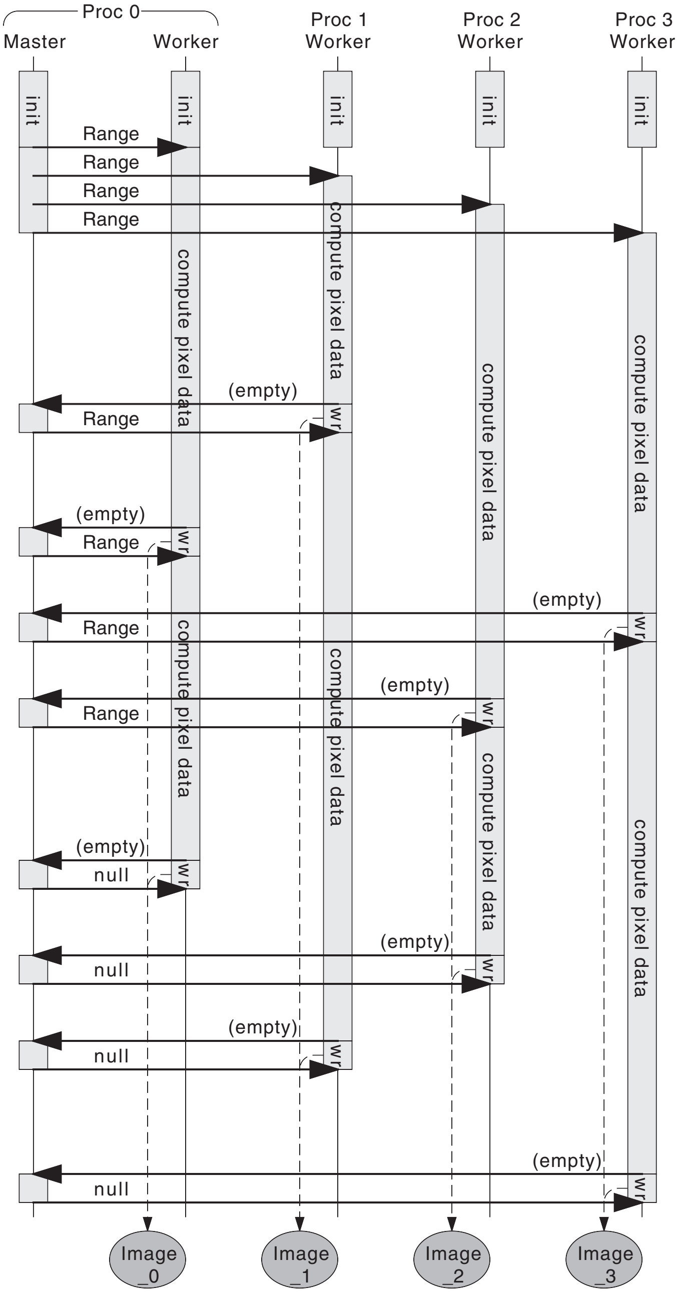


**Figure 24.4** Master-worker program execution timeline with reduced message passing

For this design to work, the image file format must support writing the image in multiple chunks, where the chunks may be noncontiguous, and not all chunks of the image may be present in any one file. The PJG image file format supports images with multiple chunks scattered among separate files. The PJG image viewer program can take several PJG files containing parts of an image and display the resulting combined image. The PJG image viewer program can also store the combined image in a single PJG file if desired.

Eliminating the pixel data messages also improves the program's memory scalability. No longer does the master process need to allocate storage for the entire pixel matrix. Each process needs to allocate storage for only one slice, and can reuse this storage to compute each slice.

Furthermore, eliminating the pixel data messages improves the program's scalability in another way. In both the previous design and the new design, the computation time is $O(n^2)$ for an $n \times n$-pixel image—each pixel's value has to be computed. In the previous design, the message-passing time is also $O(n^2)$—each pixel's value has to be sent to the master. But in the new design, the message-passing time is only $O(n)$; the number of messages is proportional to the number of *rows* (not pixels) in the image, and each message carries a constant amount of data. Thus, in the new design, the ratio of computation to communication is $O(n)$. As the problem size scales up, the sequential fraction occupied by message passing should decrease and the parallel performance should improve—a manifestation of the *surface-to-volume effect.* The ratio of computation to communication in the previous design is $O(1)$, so its sequential fraction and parallel performance do not change much as the problem size scales up (as is apparent in Figure 23.10).

This new Mandelbrot Set program's design is an example of a general pattern—the **parallel output files** pattern. The program stores its results in multiple output files in parallel, rather than in one output file sequentially; the multiple output files are combined or post-processed later. The parallel output files pattern's benefits are typically improved performance and scalability. In later chapters, we will see additional examples of the parallel output files pattern.

## 24.5 Program with Reduced Message Passing

Here is the source code for class MandelbrotSetClu3 using the master-worker pattern for load balancing along with the parallel output files pattern.

```
package edu.rit.clu.fractal;
import edu.rit.color.HSB;
import edu.rit.image.PJGColorImage;
import edu.rit.image.PJGImage;
import edu.rit.io.Files;
import edu.rit.mp.IntegerBuf;
import edu.rit.mp.ObjectBuf;
import edu.rit.mp.buf.ObjectItemBuf;
import edu.rit.pj.Comm;
import edu.rit.pj.CommRequest;
import edu.rit.pj.CommStatus;
import edu.rit.pj.IntegerSchedule;
import edu.rit.pj.ParallelRegion;
import edu.rit.pj.ParallelSection;
import edu.rit.pj.ParallelTeam;
import edu.rit.util.Range;
import java.io.BufferedOutputStream;
import java.io.File;
import java.io.FileOutputStream;
import java.io.IOException;
public class MandelbrotSetClu3
   {
   // Communicator.
   static Comm world;
   static int size;
   static int rank;

   // Command line arguments.
   static int width;
   static int height;
   static double xcenter;
   static double ycenter;
   static double resolution;
   static int maxiter;
   static double gamma;
   static File filename;

   // Initial pixel offsets from center.
   static int xoffset;
   static int yoffset;

   // Image matrix.
   static int[][] matrix;
   static PJGColorImage image;
   static PJGImage.Writer writer;

   // Table of hues.
   static int[] huetable;

   // Message tags.
   static final int WORKER_MSG = 0;
   static final int MASTER_MSG = 1;

   // Number of chunks the worker computed.
   static int chunkCount;

   /**
    * Mandelbrot Set main program.
    */
   public static void main
```

```
         (String[] args)
         throws Exception
         {
         // Start timing.
         long t1 = System.currentTimeMillis();

         // Initialize middleware.
         Comm.init (args);
         world = Comm.world();
         size = world.size();
         rank = world.rank();

         // Validate command line arguments.
         if (args.length != 8) usage();
         width = Integer.parseInt (args[0]);
         height = Integer.parseInt (args[1]);
         xcenter = Double.parseDouble (args[2]);
         ycenter = Double.parseDouble (args[3]);
         resolution = Double.parseDouble (args[4]);
         maxiter = Integer.parseInt (args[5]);
         gamma = Double.parseDouble (args[6]);
         filename = new File (args[7]);

         // Initial pixel offsets from center.
         xoffset = -(width - 1) / 2;
         yoffset = (height - 1) / 2;
```

To write the worker's PJG image file, we still need a pixel data matrix of type `int[][]`. However, because we will never write the entire image, we don't need to allocate storage for both dimensions of the matrix; we need to allocate storage only for the first dimension (the row references).

```
         // Allocate storage for pixel matrix row references only.
         matrix = new int [height] [];
```

Each process sets up a writer for its own PJG image file. The `Files.fileforRank()` method takes the file name from the command line and appends the process's rank. For example, if the file name is specified on the command line as `image.pjg`, the workers will write files named `image_0.pjg`, `image_1.pjg`, and so on.

```
         // Prepare to write image row slices to per-worker PJG image
         // file.
         image = new PJGColorImage (height, width, matrix);
         writer =
            image.prepareToWrite
               (new BufferedOutputStream
                  (new FileOutputStream
                     (Files.fileForRank (filename, rank))));

         // Create table of hues for different iteration counts.
         huetable = new int [maxiter+1];
         for (int i = 0; i < maxiter; ++ i)
            {
            huetable[i] = HSB.pack
               (/*hue*/ (float)
                  Math.pow (((double)i)/((double)maxiter),gamma),
                /*sat*/ 1.0f,
                /*bri*/ 1.0f);
            }
         huetable[maxiter] = HSB.pack (1.0f, 1.0f, 0.0f);

         long t2 = System.currentTimeMillis();

         // In master process, run master section and worker section
         // in parallel.
         if (rank == 0)
            {
            new ParallelTeam(2).execute (new ParallelRegion()
               {
               public void run() throws Exception
                  {
                  execute (new ParallelSection()
                     {
                     public void run() throws Exception
                        {
                        masterSection();
                        }
                     },
                  new ParallelSection()
                     {
                     public void run() throws Exception
                        {
                        workerSection();
                        }
                     });
                  }
               });
            }

         // In worker process, run only worker section.
         else
            {
            workerSection();
            }

         long t3 = System.currentTimeMillis();

         // Close image file.
         writer.close();

         // Stop timing.
         long t4 = System.currentTimeMillis();
         System.out.println (chunkCount + " chunks " + rank);
         System.out.println ((t2-t1) + " msec pre " + rank);
         System.out.println ((t3-t2) + " msec calc " + rank);
         System.out.println ((t4-t3) + " msec post " + rank);
         System.out.println ((t4-t1) + " msec total " + rank);
         }
```

The master section is similar to the one in the MandelbrotSetClu2 program, except the master receives an empty message from any worker to detect when a worker has finished computing a slice, and the master does not receive pixel data from the worker.

```
/**
 * Perform the master section.
 */
private static void masterSection()
   throws IOException
   {
   int worker;
   Range range;

   // Set up a schedule object to divide the row range into
   // chunks.
   IntegerSchedule schedule = IntegerSchedule.runtime();
   schedule.start (size, new Range (0, height-1));

   // Send initial chunk range to each worker. If range is null,
   // no more work for that worker. Keep count of active workers.
   int activeWorkers = size;
   for (worker = 0; worker < size; ++ worker)
      {
      range = schedule.next (worker);
      world.send (worker, WORKER_MSG, ObjectBuf.buffer (range));
      if (range == null) -- activeWorkers;
      }
   // Repeat until all workers have finished.
   while (activeWorkers > 0)
      {
      // Receive an empty message from any worker.
      CommStatus status =
         world.receive
            (null, MASTER_MSG, IntegerBuf.emptyBuffer());
      worker = status.fromRank;

      // Send next chunk range to that specific worker. If null,
      // no more work.
      range = schedule.next (worker);
      world.send (worker, WORKER_MSG, ObjectBuf.buffer (range));
      if (range == null) -- activeWorkers;
      }
   }
```

The worker section starts out the same as the one in the MandelbrotSetClu2 program.

```
   /**
    * Perform the worker section.
    */
   private static void workerSection()
      throws IOException
      {
      // Storage for matrix row slice.
      int[][] slice = null;

      // Process chunks from master.
      for (;;)
         {
         // Receive chunk range from master. If null, no more work.
         ObjectItemBuf<Range> rangeBuf = ObjectBuf.buffer();
         world.receive (0, WORKER_MSG, rangeBuf);
         Range range = rangeBuf.item;
         if (range == null) break;
         int lb = range.lb();
         int ub = range.ub();
         int len = range.length();
         ++ chunkCount;

         // Allocate storage for matrix row slice if necessary.
         if (slice == null || slice.length < len)
            {
```

```
   slice = new int [len] [width];
   }

// Compute all rows and columns in slice.
for (int r = lb; r <= ub; ++ r)
   {
   int[] slice_r = slice[r-lb];
   double y = ycenter + (yoffset - r) / resolution;

   for (int c = 0; c < width; ++ c)
      {
      double x = xcenter + (xoffset + c) / resolution;

      // Iterate until convergence.
      int i = 0;
      double aold = 0.0;
      double bold = 0.0;
      double a = 0.0;
      double b = 0.0;
      double zmagsqr = 0.0;
      while (i < maxiter && zmagsqr <= 4.0)
         {
         ++ i;
         a = aold*aold - bold*bold + x;
         b = 2.0*aold*bold + y;
         zmagsqr = a*a + b*b;
         aold = a;
         bold = b;
         }

      // Record number of iterations for pixel.
      slice_r[c] = huetable[i];
      }
   }

// Report completion of slice to master.
world.send (0, MASTER_MSG, IntegerBuf.emptyBuffer());
```

At this point, the worker sends an empty message to the master to report completion of its computations. While this message is traversing the network and the master's reply is returning, the worker is busy writing its slice to its own PJG image file. Thus, the file I/O is *overlapped* with the message passing.

To get the proper row indexes written into the PJG image file, the just-computed slice must occupy the proper row indexes in the full pixel data matrix. We set this up by copying the row *references* from the `slice` variable to the `matrix` variable. (There's no need to copy the row *elements*.)

```
      // Set full pixel matrix rows to refer to slice rows.
      System.arraycopy (slice, 0, matrix, lb, len);

      // Write row slice of full pixel matrix to image file.
      writer.writeRowSlice (range);
      }
   }
}
```

Table 24.2 (at the end of the chapter) lists, and Figure 24.5 plots, the MandelbrotSetClu3 program's running-time metrics. Because of the surface-to-volume effect, the speedups and efficiencies become better and better as the problem size scales up. For a 640M-pixel image, the efficiencies remain above 98 percent, even as the number of processors increases to 40. Likewise, the *EDSF* values decrease as the problem size increases.

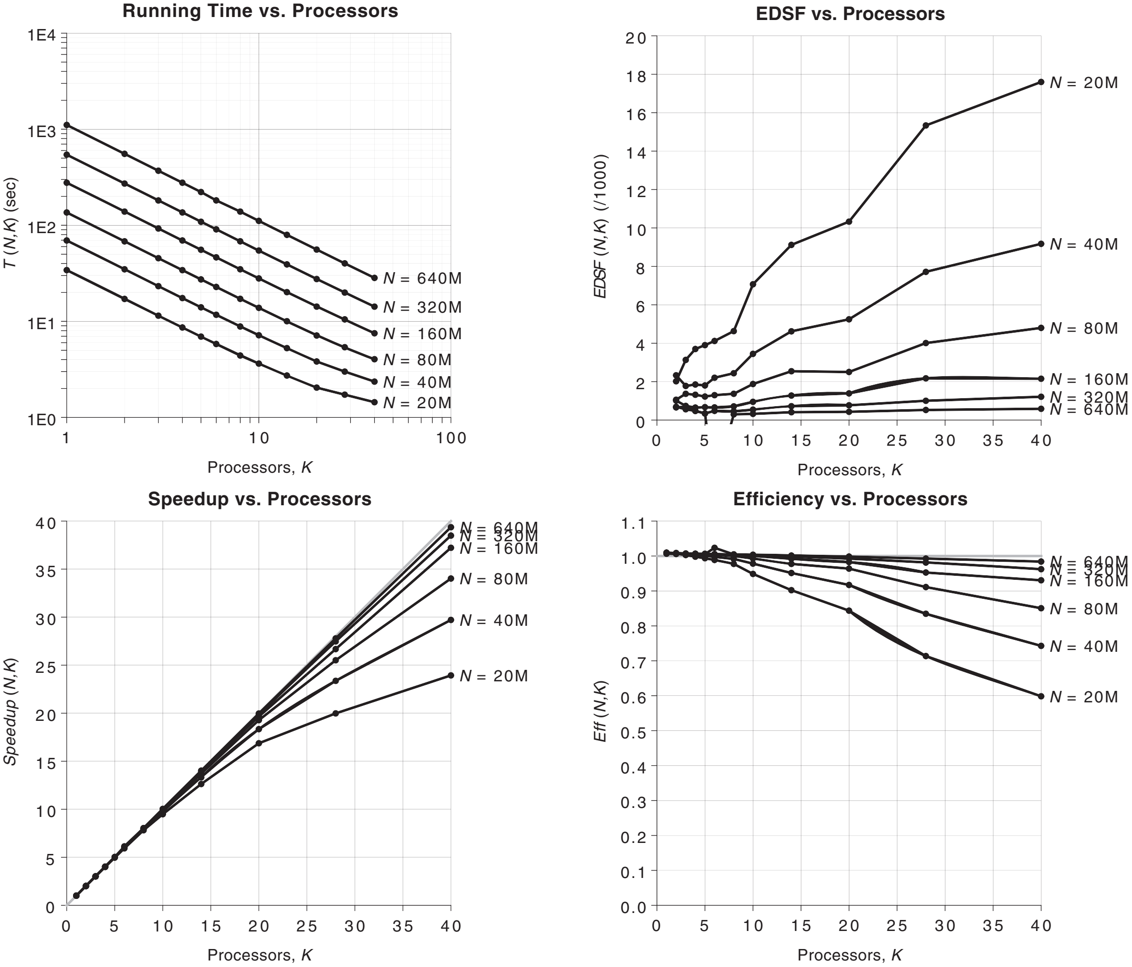


**Figure 24.5** MandelbrotSetSeq/MandelbrotSetClu3 running-time metrics

That the *EDSF* curves increase as *K* increases is due to the JIT compiler. For a small problem size and a large number of processors, the program runs only for a few seconds. This is not enough time for the JIT compiler to detect hot spots and compile them to machine code. The program thus takes longer to finish than it would otherwise, and this increases the measured *EDSFs*. For a large problem size or a small number of processors, the program runs long enough for the JIT compiler to do its optimizations, letting the program's performance achieve its full potential.

## 24.6 Message Scatter and Gather Time Models

Equation 24.1 models the time to send and receive a *point-to-point* message. This suffices to analyze the master-worker Mandelbrot Set program's running time. But we are also interested in modeling the time needed for *collective* communication operations, such as scatter and gather. To do that, we need to know how Parallel Java implements scatter and gather.

In a cluster parallel program with *K* processes, a scatter operation transfers the contents of *K* source buffers in the root process to one destination buffer in each of the *K* processes. In the root process, the scatter just copies the source buffer to the destination buffer without sending a message. (If the source buffer and the destination buffer are the same, the scatter does nothing at all in the root process.) For the other source buffers, the scatter sends a message from the root process to each destination process in sequence (Figure 24.6). Assuming each source buffer has the same number of data items, the time to do a scatter is (*K*–1) times the time to send a message. For the "tardis" computer, the **message scatter-time model** is

$$T(B,K) = (2.08\times10^{-4} + 1.07\times10^{-9}\ B)\cdot(K-1) \tag{24.2}$$

where *B* is the number of data bits in one source buffer, *K* is the number of processes, and *T* is the scatter time in seconds.

A gather is merely a scatter in reverse (Figure 24.7). Therefore, the **message gather time model** is the same as the message scatter time model.

Any message-passing library's goal is to implement collective communication operations, such as scatter and gather, so as to minimize the communication time. However, the message pattern that minimizes the communication time might be platform dependent; in particular, it might depend on the cluster parallel computer's backend network interconnection technology. Thus, different message passing libraries might implement the same collective communication operation in different ways.

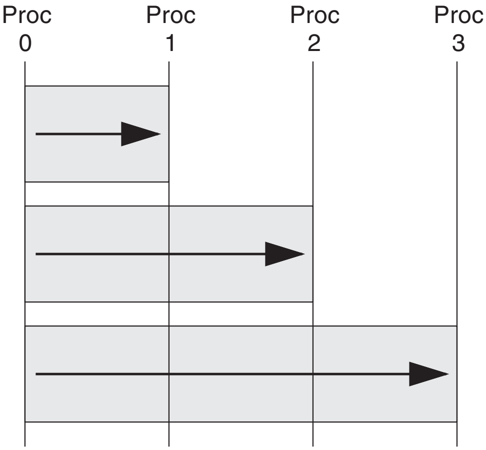


**Figure 24.6** Messages for a scatter

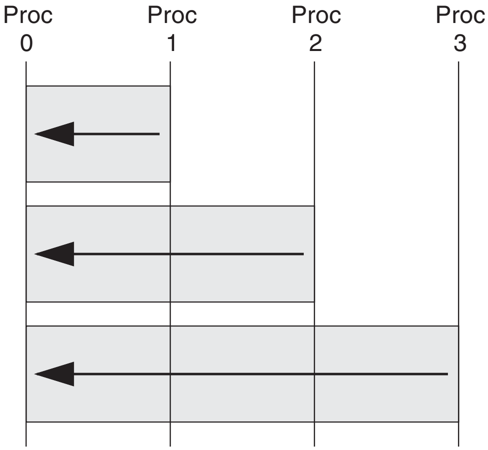


**Figure 24.7** Messages for a gather

The Parallel Java Library is targeted primarily for commodity clusters, in which each backend node has one Ethernet network interface. With that kind of interconnect, the individual messages in the scatter or gather have to traverse the source or destination node's single network interface one at a time, leading to the message patterns in Figures 24.6 and 24.7.

Alternatively, a cluster parallel computer using an interconnection technology other than commodity off-the-shelf Ethernet might have more than one network interface on each node. A platform-specific message-passing library targeted for that interconnect can then send some or all of the messages simultaneously, thus reducing the communication time.

Consequently, any collective communication operation's message-time model is implementation dependent. Equation 24.2 pertains specifically to the Parallel Java implementation of scatter and gather. Message-time models for other collective communication operations discussed in later chapters will likewise pertain specifically to the Parallel Java implementation. When analyzing the communication time of a cluster parallel program with collective communication operations, be sure to use message-time models that correspond to the message-passing library implementation you are using.

## 24.7 Intra-Node Message Passing

When a process at one rank sends a message to a process at a different rank in a cluster parallel program, we've tacitly assumed that the destination process is located in a backend node different from the source process, and that the message must therefore go across the backend network. The data used to derive the message send-time model (Table 24.1) was in fact obtained by running the two ping-pong processes on different backend nodes. Equation 24.1 thus represents an **inter-node message send-time model** for the "tardis" computer.

On a regular cluster parallel computer, where each backend node has just one CPU, every process automatically runs on a different backend node. But on a hybrid parallel computer, it is possible to run more than one process on the same node. Figure 24.8 shows how the processes of the Mandelbrot Set program were assigned to the ten nodes of the "tardis" computer for *K*=10 and *K*=14 processes. For *K*=10 (and below), each process was assigned to a different node, and the program used only one of the four CPUs on each node. For *K*=14 (and *K*=20), two processes were assigned to each node, and the program used two of the four CPUs on each node. For *K*=28 and *K*=40, four processes were assigned to each node, and the program used all four CPUs on each node.

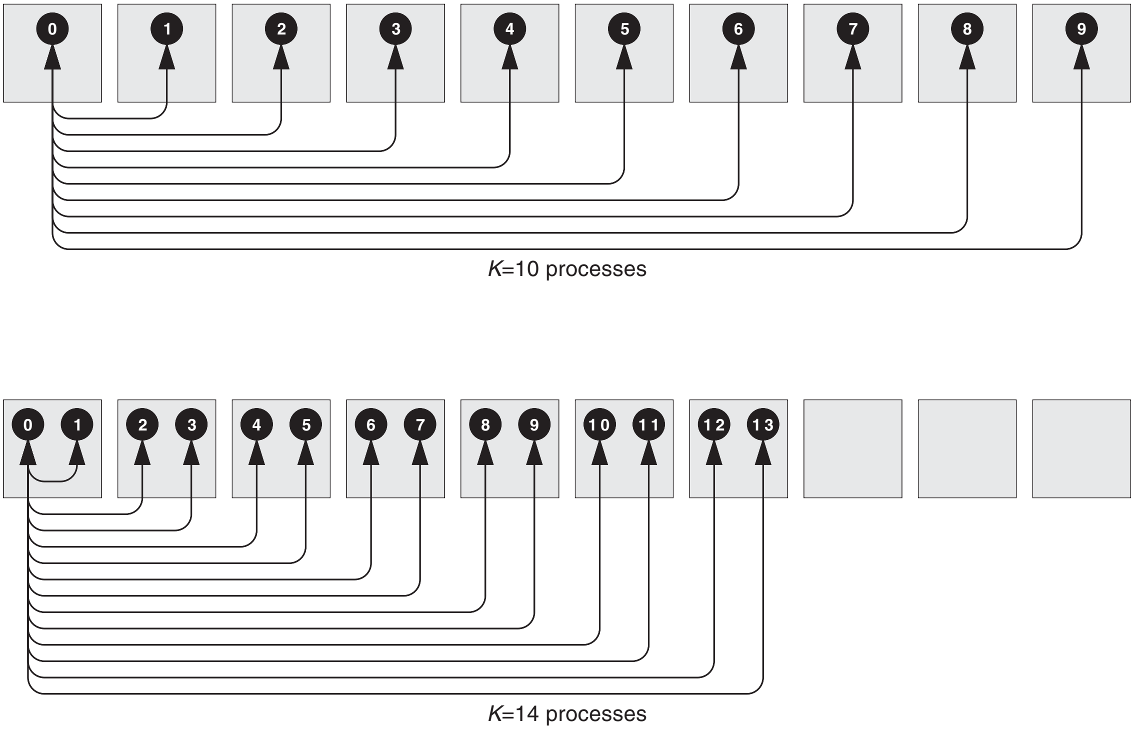


**Figure 24.8** Mandelbrot Set program process allocation and message passing on the "tardis" computer

Figure 24.8 also shows the messages sent back and forth between the master process (rank 0) and the worker processes (ranks 1 through *K*–1). For *K*=10 and below, all inter-process messages are also inter-node messages that travel over the backend network. (We are not considering messages a process sends to itself, such as messages between the master and the worker in process 0.) But for *K*=14 and above, some inter-process messages are **intra-node messages**—messages between different processes on the same node. These messages do not travel over the backend network; rather, they go from the source process to the node's operating system kernel, and from there to the destination process. Thus, the inter-node message send-time model (Equation 24.1) does not apply to these intra-node messages.

We need a model for the time needed to send a message between different processes on the same backend node. Table 24.3 (at the end of the chapter) gives the measurements from the TimeSendByte, TimeSendInt, and TimeSendDouble programs running with two processes on the same backend node. A linear regression yields the **intra-node message send-time model** for the “tardis” computer,

$$T(B) = 7.89 \times 10^{-5} + 2.26 \times 10^{-10}\, B \tag{24.3}$$

where *T* is the message send time in seconds and *B* is the message data size in bits. The correlation coefficient is 0.998666. Compared to the inter-node message send-time model, note the smaller latency (78.9 $\mu$sec instead of 208 $\mu$sec) and higher bandwidth (4.423 Gbps instead of 0.937 Gbps) when sending messages between processes in the same node.

We will use both the inter-node and intra-node message send-time models again in later chapters when we analyze the running times of our cluster parallel programs.

**Table 24.1** Message send-time data for the “tardis” computer, inter-node

| *Bytes* | *N (bits)* | *T (sec)* | *Ints* | *N (bits)* | *T (sec)* | *Doubles* | *N (bits)* | *T (sec)* |
|---:|---:|---:|---:|---:|---:|---:|---:|---:|
| 100 | 800 | 1.15E-4 | 100 | 3200 | 1.34E-4 | 100 | 6400 | 1.50E-4 |
| 100 | 800 | 1.34E-4 | 100 | 3200 | 1.39E-4 | 100 | 6400 | 1.37E-4 |
| 100 | 800 | 1.34E-4 | 100 | 3200 | 1.34E-4 | 100 | 6400 | 1.36E-4 |
| 200 | 1600 | 1.34E-4 | 200 | 6400 | 1.35E-4 | 200 | 12800 | 1.69E-4 |
| 200 | 1600 | 1.15E-4 | 200 | 6400 | 1.34E-4 | 200 | 12800 | 1.99E-4 |
| 200 | 1600 | 8.70E-5 | 200 | 6400 | 1.34E-4 | 200 | 12800 | 2.01E-4 |
| 500 | 4000 | 1.35E-4 | 500 | 16000 | 1.50E-4 | 500 | 32000 | 2.01E-4 |
| 500 | 4000 | 1.33E-4 | 500 | 16000 | 2.01E-4 | 500 | 32000 | 2.00E-4 |
| 500 | 4000 | 1.34E-4 | 500 | 16000 | 2.01E-4 | 500 | 32000 | 1.95E-4 |
| 1000 | 8000 | 1.36E-4 | 1000 | 32000 | 1.90E-4 | 1000 | 64000 | 2.67E-4 |
| 1000 | 8000 | 1.45E-4 | 1000 | 32000 | 1.77E-4 | 1000 | 64000 | 2.57E-4 |
| 1000 | 8000 | 1.58E-4 | 1000 | 32000 | 1.87E-4 | 1000 | 64000 | 2.60E-4 |
| 2000 | 16000 | 1.62E-4 | 2000 | 64000 | 2.33E-4 | 2000 | 128000 | 3.33E-4 |
| 2000 | 16000 | 1.35E-4 | 2000 | 64000 | 2.34E-4 | 2000 | 128000 | 3.35E-4 |
| 2000 | 16000 | 1.79E-4 | 2000 | 64000 | 2.68E-4 | 2000 | 128000 | 3.35E-4 |
| 5000 | 40000 | 2.00E-4 | 5000 | 160000 | 3.81E-4 | 5000 | 320000 | 5.62E-4 |
| 5000 | 40000 | 2.14E-4 | 5000 | 160000 | 3.80E-4 | 5000 | 320000 | 5.66E-4 |
| 5000 | 40000 | 2.06E-4 | 5000 | 160000 | 3.80E-4 | 5000 | 320000 | 5.64E-4 |
| 10000 | 80000 | 2.64E-4 | 10000 | 320000 | 5.82E-4 | 10000 | 640000 | 8.85E-4 |
| 10000 | 80000 | 2.64E-4 | 10000 | 320000 | 5.83E-4 | 10000 | 640000 | 8.88E-4 |
| 10000 | 80000 | 2.64E-4 | 10000 | 320000 | 5.84E-4 | 10000 | 640000 | 8.83E-4 |
| 20000 | 160000 | 3.74E-4 | 20000 | 640000 | 9.12E-4 | 20000 | 1280000 | 1.60E-3 |
| 20000 | 160000 | 3.72E-4 | 20000 | 640000 | 9.00E-4 | 20000 | 1280000 | 1.61E-3 |
| 20000 | 160000 | 3.73E-4 | 20000 | 640000 | 9.12E-4 | 20000 | 1280000 | 1.61E-3 |
| 50000 | 400000 | 6.52E-4 | 50000 | 1600000 | 1.98E-3 | 50000 | 3200000 | 3.67E-3 |
| 50000 | 400000 | 6.52E-4 | 50000 | 1600000 | 1.97E-3 | 50000 | 3200000 | 3.69E-3 |
| 50000 | 400000 | 6.52E-4 | 50000 | 1600000 | 1.97E-3 | 50000 | 3200000 | 3.69E-3 |
| 100000 | 800000 | 1.08E-3 | 100000 | 3200000 | 3.73E-3 | 100000 | 6400000 | 7.19E-3 |
| 100000 | 800000 | 1.08E-3 | 100000 | 3200000 | 3.73E-3 | 100000 | 6400000 | 7.17E-3 |
| 100000 | 800000 | 1.08E-3 | 100000 | 3200000 | 3.73E-3 | 100000 | 6400000 | 7.19E-3 |
| 200000 | 1600000 | 1.95E-3 | 200000 | 6400000 | 7.21E-3 | 200000 | 12800000 | 1.39E-2 |
| 200000 | 1600000 | 1.95E-3 | 200000 | 6400000 | 7.21E-3 | 200000 | 12800000 | 1.38E-2 |
| 200000 | 1600000 | 1.95E-3 | 200000 | 6400000 | 7.20E-3 | 200000 | 12800000 | 1.40E-2 |
| 500000 | 4000000 | 4.50E-3 | 500000 | 16000000 | 1.76E-2 | 500000 | 32000000 | 3.41E-2 |
| 500000 | 4000000 | 4.49E-3 | 500000 | 16000000 | 1.77E-2 | 500000 | 32000000 | 3.42E-2 |
| 500000 | 4000000 | 4.50E-3 | 500000 | 16000000 | 1.75E-2 | 500000 | 32000000 | 3.42E-2 |
| 1000000 | 8000000 | 8.82E-3 | 1000000 | 32000000 | 3.46E-2 | 1000000 | 64000000 | 6.84E-2 |
| 1000000 | 8000000 | 8.82E-3 | 1000000 | 32000000 | 3.43E-2 | 1000000 | 64000000 | 6.85E-2 |
| 1000000 | 8000000 | 8.82E-3 | 1000000 | 32000000 | 3.47E-2 | 1000000 | 64000000 | 6.84E-2 |

**Table 24.2** MandelbrotSetSeq/MandelbrotSetClu3 running-time metrics

| *N* | *K* | *T* | *Spdup* | *Eff* | *EDSF* | *N* | *K* | *T* | *Spdup* | *Eff* | *EDSF* |
|---:|---:|---:|---:|---:|---:|---:|---:|---:|---:|---:|---:|
| 20M | seq | 34468 | | | | 160M | seq | 279753 | | | |
| 20M | 1 | 34157 | 1.009 | 1.009 | | 160M | 1 | 277358 | 1.009 | 1.009 | |
| 20M | 2 | 17113 | 2.014 | 1.007 | 0.002 | 160M | 2 | 138820 | 2.015 | 1.008 | 0.001 |
| 20M | 3 | 11457 | 3.008 | 1.003 | 0.003 | 160M | 3 | 92589 | 3.021 | 1.007 | 0.001 |
| 20M | 4 | 8634 | 3.992 | 0.998 | 0.004 | 160M | 4 | 69475 | 4.027 | 1.007 | 0.001 |
| 20M | 5 | 6938 | 4.968 | 0.994 | 0.004 | 160M | 5 | 55621 | 5.030 | 1.006 | 0.001 |
| 20M | 6 | 5810 | 5.933 | 0.989 | 0.004 | 160M | 6 | 46379 | 6.032 | 1.005 | 0.001 |
| 20M | 8 | 4408 | 7.819 | 0.977 | 0.005 | 160M | 8 | 34843 | 8.029 | 1.004 | 0.001 |
| 20M | 10 | 3633 | 9.487 | 0.949 | 0.007 | 160M | 10 | 27973 | 10.001 | 1.000 | 0.001 |
| 20M | 14 | 2729 | 12.630 | 0.902 | 0.009 | 160M | 14 | 20140 | 13.890 | 0.992 | 0.001 |
| 20M | 20 | 2043 | 16.871 | 0.844 | 0.010 | 160M | 20 | 14234 | 19.654 | 0.983 | 0.001 |
| 20M | 28 | 1725 | 19.981 | 0.714 | 0.015 | 160M | 28 | 10487 | 26.676 | 0.953 | 0.002 |
| 20M | 40 | 1440 | 23.936 | 0.598 | 0.018 | 160M | 40 | 7516 | 37.221 | 0.931 | 0.002 |
| 40M | seq | 70089 | | | | 320M | seq | 547530 | | | |
| 40M | 1 | 69494 | 1.009 | 1.009 | | 320M | 1 | 543269 | 1.008 | 1.008 | |
| 40M | 2 | 34828 | 2.012 | 1.006 | 0.002 | 320M | 2 | 271814 | 2.014 | 1.007 | 0.001 |
| 40M | 3 | 23247 | 3.015 | 1.005 | 0.002 | 320M | 3 | 181341 | 3.019 | 1.006 | 0.001 |
| 40M | 4 | 17470 | 4.012 | 1.003 | 0.002 | 320M | 4 | 136012 | 4.026 | 1.006 | 0.000 |
| 40M | 5 | 13999 | 5.007 | 1.001 | 0.002 | 320M | 5 | 108803 | 5.032 | 1.006 | 0.000 |
| 40M | 6 | 11710 | 5.985 | 0.998 | 0.002 | 320M | 6 | 90762 | 6.033 | 1.005 | 0.000 |
| 40M | 8 | 8835 | 7.933 | 0.992 | 0.002 | 320M | 8 | 68127 | 8.037 | 1.005 | 0.000 |
| 40M | 10 | 7165 | 9.782 | 0.978 | 0.003 | 320M | 10 | 54593 | 10.029 | 1.003 | 0.001 |
| 40M | 14 | 5262 | 13.320 | 0.951 | 0.005 | 320M | 14 | 39169 | 13.979 | 0.998 | 0.001 |
| 40M | 20 | 3821 | 18.343 | 0.917 | 0.005 | 320M | 20 | 27561 | 19.866 | 0.993 | 0.001 |
| 40M | 28 | 2999 | 23.371 | 0.835 | 0.008 | 320M | 28 | 19929 | 27.474 | 0.981 | 0.001 |
| 40M | 40 | 2359 | 29.711 | 0.743 | 0.009 | 320M | 40 | 14225 | 38.491 | 0.962 | 0.001 |
| 80M | seq | 137280 | | | | 640M | seq | 1115408 | | | |
| 80M | 1 | 135954 | 1.010 | 1.010 | | 640M | 1 | 1108042 | 1.007 | 1.007 | |
| 80M | 2 | 68049 | 2.017 | 1.009 | 0.001 | 640M | 2 | 554404 | 2.012 | 1.006 | 0.001 |
| 80M | 3 | 45442 | 3.021 | 1.007 | 0.001 | 640M | 3 | 369763 | 3.017 | 1.006 | 0.001 |
| 80M | 4 | 34123 | 4.023 | 1.006 | 0.001 | 640M | 4 | 277406 | 4.021 | 1.005 | 0.000 |
| 80M | 5 | 27324 | 5.024 | 1.005 | 0.001 | 640M | 5 | 221917 | 5.026 | 1.005 | 0.000 |
| 80M | 6 | 22806 | 6.019 | 1.003 | 0.001 | 640M | 6 | 181632 | 6.141 | 1.024 | -0.003 |
| 80M | 8 | 17157 | 8.001 | 1.000 | 0.001 | 640M | 8 | 138794 | 8.036 | 1.005 | 0.000 |
| 80M | 10 | 13825 | 9.930 | 0.993 | 0.002 | 640M | 10 | 111127 | 10.037 | 1.004 | 0.000 |
| 80M | 14 | 10032 | 13.684 | 0.977 | 0.003 | 640M | 14 | 79567 | 14.018 | 1.001 | 0.000 |
| 80M | 20 | 7121 | 19.278 | 0.964 | 0.003 | 640M | 20 | 55856 | 19.969 | 0.998 | 0.000 |
| 80M | 28 | 5381 | 25.512 | 0.911 | 0.004 | 640M | 28 | 40136 | 27.791 | 0.993 | 0.001 |
| 80M | 40 | 4035 | 34.022 | 0.851 | 0.005 | 640M | 40 | 28336 | 39.364 | 0.984 | 0.001 |

**Table 24.3** Message send-time data for the "tardis" computer, intra-node

| *Bytes* | *N (bits)* | *T (sec)* | *Ints* | *N (bits)* | *T (sec)* | *Doubles* | *N (bits)* | *T (sec)* |
|---:|---:|---:|---:|---:|---:|---:|---:|---:|
| 100 | 800 | 3.36E-5 | 100 | 3200 | 3.47E-5 | 100 | 6400 | 3.56E-5 |
| 100 | 800 | 3.30E-5 | 100 | 3200 | 3.50E-5 | 100 | 6400 | 3.54E-5 |
| 100 | 800 | 3.28E-5 | 100 | 3200 | 3.54E-5 | 100 | 6400 | 3.66E-5 |
| 200 | 1600 | 3.26E-5 | 200 | 6400 | 3.66E-5 | 200 | 12800 | 3.90E-5 |
| 200 | 1600 | 3.26E-5 | 200 | 6400 | 3.67E-5 | 200 | 12800 | 3.79E-5 |
| 200 | 1600 | 3.45E-5 | 200 | 6400 | 3.62E-5 | 200 | 12800 | 3.88E-5 |
| 500 | 4000 | 3.35E-5 | 500 | 16000 | 4.02E-5 | 500 | 32000 | 4.78E-5 |
| 500 | 4000 | 3.40E-5 | 500 | 16000 | 4.00E-5 | 500 | 32000 | 4.76E-5 |
| 500 | 4000 | 3.40E-5 | 500 | 16000 | 3.91E-5 | 500 | 32000 | 4.68E-5 |
| 1000 | 8000 | 3.52E-5 | 1000 | 32000 | 4.58E-5 | 1000 | 64000 | 6.21E-5 |
| 1000 | 8000 | 3.60E-5 | 1000 | 32000 | 4.73E-5 | 1000 | 64000 | 6.35E-5 |
| 1000 | 8000 | 3.70E-5 | 1000 | 32000 | 4.58E-5 | 1000 | 64000 | 6.17E-5 |
| 2000 | 16000 | 3.74E-5 | 2000 | 64000 | 6.06E-5 | 2000 | 128000 | 9.18E-5 |
| 2000 | 16000 | 3.74E-5 | 2000 | 64000 | 6.06E-5 | 2000 | 128000 | 8.94E-5 |
| 2000 | 16000 | 3.68E-5 | 2000 | 64000 | 6.12E-5 | 2000 | 128000 | 9.32E-5 |
| 5000 | 40000 | 4.49E-5 | 5000 | 160000 | 1.02E-4 | 5000 | 320000 | 1.65E-4 |
| 5000 | 40000 | 4.68E-5 | 5000 | 160000 | 1.03E-4 | 5000 | 320000 | 1.68E-4 |
| 5000 | 40000 | 4.54E-5 | 5000 | 160000 | 1.04E-4 | 5000 | 320000 | 1.72E-4 |
| 10000 | 80000 | 5.73E-5 | 10000 | 320000 | 1.64E-4 | 10000 | 640000 | 2.54E-4 |
| 10000 | 80000 | 5.72E-5 | 10000 | 320000 | 1.62E-4 | 10000 | 640000 | 2.64E-4 |
| 10000 | 80000 | 5.58E-5 | 10000 | 320000 | 1.64E-4 | 10000 | 640000 | 2.59E-4 |
| 20000 | 160000 | 8.28E-5 | 20000 | 640000 | 2.55E-4 | 20000 | 1280000 | 4.50E-4 |
| 20000 | 160000 | 8.60E-5 | 20000 | 640000 | 2.52E-4 | 20000 | 1280000 | 4.28E-4 |
| 20000 | 160000 | 8.13E-5 | 20000 | 640000 | 2.66E-4 | 20000 | 1280000 | 4.43E-4 |
| 50000 | 400000 | 1.45E-4 | 50000 | 1600000 | 5.30E-4 | 50000 | 3200000 | 9.94E-4 |
| 50000 | 400000 | 1.46E-4 | 50000 | 1600000 | 5.26E-4 | 50000 | 3200000 | 9.72E-4 |
| 50000 | 400000 | 1.48E-4 | 50000 | 1600000 | 5.61E-4 | 50000 | 3200000 | 9.69E-4 |
| 100000 | 800000 | 2.46E-4 | 100000 | 3200000 | 9.95E-4 | 100000 | 6400000 | 1.61E-3 |
| 100000 | 800000 | 2.41E-4 | 100000 | 3200000 | 1.01E-3 | 100000 | 6400000 | 1.63E-3 |
| 100000 | 800000 | 2.42E-4 | 100000 | 3200000 | 9.80E-4 | 100000 | 6400000 | 1.64E-3 |
| 200000 | 1600000 | 4.29E-4 | 200000 | 6400000 | 1.62E-3 | 200000 | 12800000 | 2.93E-3 |
| 200000 | 1600000 | 4.34E-4 | 200000 | 6400000 | 1.64E-3 | 200000 | 12800000 | 2.83E-3 |
| 200000 | 1600000 | 4.35E-4 | 200000 | 6400000 | 1.63E-3 | 200000 | 12800000 | 2.88E-3 |
| 500000 | 4000000 | 1.02E-3 | 500000 | 16000000 | 3.60E-3 | 500000 | 32000000 | 7.44E-3 |
| 500000 | 4000000 | 1.00E-3 | 500000 | 16000000 | 3.54E-3 | 500000 | 32000000 | 6.92E-3 |
| 500000 | 4000000 | 1.01E-3 | 500000 | 16000000 | 3.85E-3 | 500000 | 32000000 | 6.82E-3 |
| 1000000 | 8000000 | 1.84E-3 | 1000000 | 32000000 | 7.76E-3 | 1000000 | 64000000 | 1.50E-2 |
| 1000000 | 8000000 | 1.84E-3 | 1000000 | 32000000 | 7.83E-3 | 1000000 | 64000000 | 1.45E-2 |
| 1000000 | 8000000 | 1.80E-3 | 1000000 | 32000000 | 7.76E-3 | 1000000 | 64000000 | 1.38E-2 |

# CHAPTER 25

# Broadcast

in which we design a cluster parallel program that requires broadcasting messages; we learn how a broadcast is performed; we derive a mathematical model for the program's computation plus communication time; and we consider the implications for cluster parallel program design

# CHAPTER 25 Broadcast

## 25.1 Floyd's Algorithm on a Cluster

Recall the **all-pairs shortest-paths problem** from Chapter 16. We are given an input *n*×*n* distance matrix *d* representing a graph with *n* vertices, and we are to compute an output distance matrix giving the length of the shortest path between each pair of vertices using Floyd's Algorithm:

```
for $i = 0$ to $n-1$
    for $r = 0$ to $n-1$
        for $c = 0$ to $n-1$
            $d_{rc} \leftarrow \min(d_{rc}, d_{ri} + d_{ic})$
```

We can design the cluster parallel version of Floyd's Algorithm the same way we designed the first cluster parallel version of the Mandelbrot Set program in Chapter 23 (class edu.rit.clu.fractal.MandelbrotSetClu). The distance matrix is divided into row slices. Each process in the parallel program allocates storage for, and computes, its own slice. This time, the initial distance matrix comes from a file. One process, say process 0, reads the input file and *scatters* the row slices to all the processes. After running Floyd's Algorithm, the row slices are *gathered* back into process 0, which writes the result into the output file. Figure 25.1 shows this design's execution timeline.

However, as we saw in Chapter 24, this design has too much message passing. We can use the *parallel output files pattern* to eliminate the final gather and reduce the program's running time. Each process writes its slice of the final distance matrix into a separate output file in parallel with the other processes; these partial output files can later be combined into a single complete output file, if desired.

Furthermore, there's no need to do the initial scatter. We can use the **parallel input file pattern**—the counterpart of the parallel output files pattern, but on the input side. Each process reads *only its own slice* of the initial distance matrix from the input file in parallel with the other processes. To do so, each process must *skip over* the unneeded distance matrix elements and commence reading elements at the proper position in the input file. Skipping data can be done efficiently; most operating systems let you "seek" directly to a given position in a file without reading all the intervening data. Class edu.rit.io.DoubleMatrixFile provides operations for reading slices of a matrix from a file as well as for reading the entire matrix.

Figure 25.2 shows the overall execution timeline of the cluster parallel Floyd's Algorithm program. This timeline assumes all the backend nodes can access the input file—as is possible, for example, if the input file is stored on a shared file server. If the nodes cannot access shared file storage, the input file must be copied to each node before the program starts.

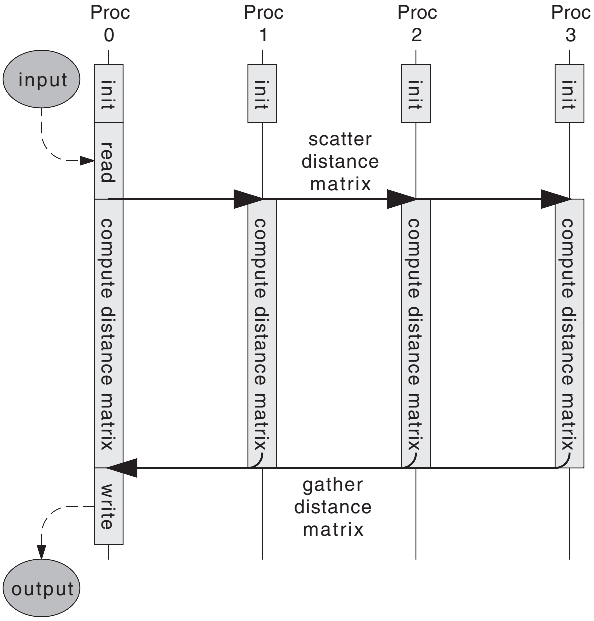


**Figure 25.1** Parallel program execution timeline with scatter/gather

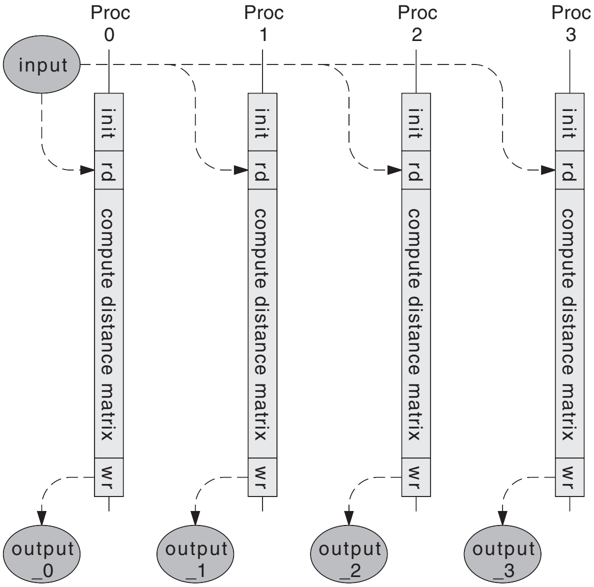


**Figure 25.2** Parallel program execution timeline with parallel input file and parallel output files

## 25.2 Collective Communication: Broadcast

Now that we've dealt with reading the input and writing the output, let's turn our attention to the computational heart of Floyd's Algorithm.

Suppose we are computing a small, 8×8-element distance matrix (`d`) in a cluster parallel program with four processes (Figure 25.3). Process 0 has rows 0–1; process 1, rows 2–3; process 2, rows 4–5; and process 3, rows 6–7. Consider what has to happen on the first outer loop iteration, where $i = 0$. Every process has to execute the following statement for all rows $r$ in the process's slice and for all columns $c$:

$$d_{rc} \leftarrow \min\left(d_{rc}, d_{r0} + d_{0c}\right)$$

That is, every process needs to access elements in row 0. This is no problem for process 0. But processes 1, 2, and 3 don't have row 0. How can they get it? Process 0 must send it to them in a message. Because *every* process needs to access row 0, process 0 sends row 0 using the *broadcast* collective communication operation. After the broadcast, all the processes execute the middle and inner loops. Then, at the beginning of the second outer loop iteration, where $i = 1$, process 0 broadcasts row 1. In general, at the beginning of outer loop iteration $i$, the process that owns row $i$ broadcasts row $i$. The cluster parallel algorithm is the following:

```
for $i = 0$ to $n-1$
    Broadcast row $i$ of $d$
    parallel for $r = 0$ to $n-1$
        for $c = 0$ to $n-1$
            $d_{rc} \leftarrow \min(d_{rc}, d_{ri} + d_{ic})$
```

The outer loop is not a parallel loop, so every process executes every outer loop iteration, beginning with the broadcast. The process that owns row $i$ acts as the root of the broadcast and sets up a source buffer referring to row $i$. The other processes set up a destination buffer for the broadcast (`row_i_buf`) that refers to an extra row's worth of storage (`row_i`). After the broadcast, each destination process uses a reference to this extra storage in lieu of a reference to (nonexistent) row $i$ in the matrix. The source process, of course, can use a reference to the actual row $i$. Figure 25.3 shows the source and destination buffers for the broadcast as well as the reference to row $i$ (`d_i`) in each process.

To execute the middle loop in a parallel fashion, the loop iterations are partitioned among the processes, just as the matrix row storage is partitioned. Each process computes its own subrange of the rows in parallel with the other processes. In the inner loop, another nonparallel loop, every process computes all the matrix columns.

Besides communicating row $i$, the broadcast operation acts as a synchronization point that enforces the sequential dependencies in Floyd's Algorithm. The broadcast operation blocks until all the processes have reached the top of the outer loop and have called the communicator's `broadcast()` method, and then the broadcast takes place and the `broadcast()` method returns. None of the processes commence the next outer loop iteration until all the processes have finished the previous outer loop iteration.

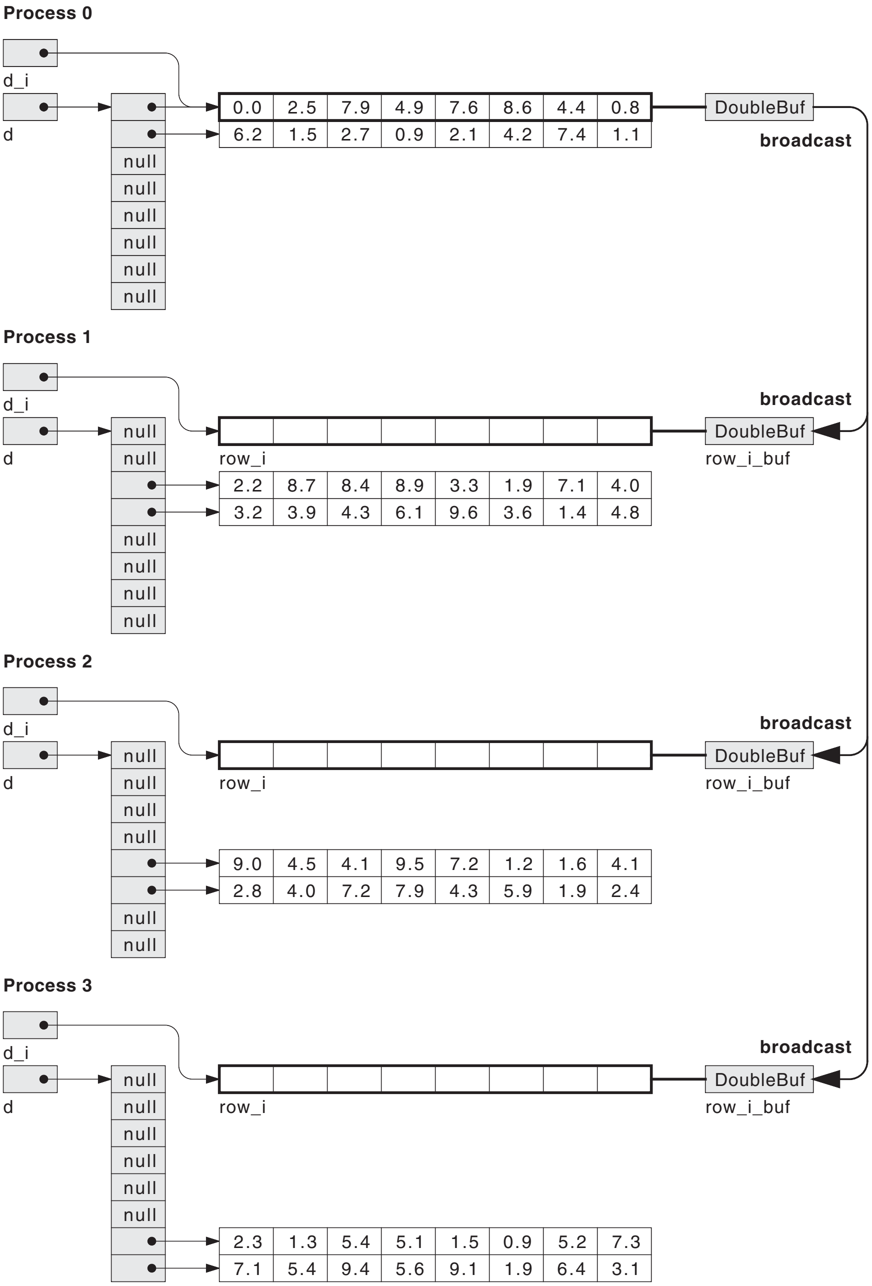


**Figure 25.3** Distance matrix sliced among *K*=4 processes, with process 0 broadcasting row 0

## 25.3 Parallel Floyd's Algorithm Program

Taking the foregoing design considerations into account, here is the code for the cluster parallel version of the Floyd's Algorithm program, class edu.rit.clu.network.FloydClu.

```
package edu.rit.clu.network;
import edu.rit.io.DoubleMatrixFile;
import edu.rit.io.Files;
import edu.rit.mp.DoubleBuf;
import edu.rit.pj.Comm;
import edu.rit.util.Range;
import java.io.BufferedInputStream;
import java.io.BufferedOutputStream;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
public class FloydClu
   {
   // World communicator.
   static Comm world;
   static int size;
   static int rank;

   // Number of nodes.
   static int n;

   // Distance matrix.
   static double[][] d;

   // Row broadcast from another process.
   static double[] row_i;
   static DoubleBuf row_i_buf;

   /**
    * Main program.
    */
   public static void main
      (String[] args)
      throws Throwable
      {
      // Start timing.
      long t1 = System.currentTimeMillis();

      // Initialize world communicator.
      Comm.init (args);
      world = Comm.world();
```

```
size = world.size();
rank = world.rank();

// Parse command line arguments.
if (args.length != 2) usage();
File infile = new File (args[0]);
File outfile = new File (args[1]);

// Prepare to read distance matrix from input file; determine
// matrix dimensions.
DoubleMatrixFile in = new DoubleMatrixFile();
DoubleMatrixFile.Reader reader =
   in.prepareToRead
      (new BufferedInputStream
         (new FileInputStream (infile)));
d = in.getMatrix();
n = d.length;

// Divide distance matrix into equal row slices.
Range[] ranges = new Range (0, n-1) .subranges (size);
Range myrange = ranges[rank];
int mylb = myrange.lb();
int myub = myrange.ub();

// Read just this process’s row slice of the distance matrix.
reader.readRowSlice (myrange);
reader.close();

// Allocate storage for row broadcast from another process.
row_i = new double [n];
row_i_buf = DoubleBuf.buffer (row_i);

long t2 = System.currentTimeMillis();

// Run Floyd’s Algorithm.
//     for i = 0 to N-1
//         for r = 0 to N-1
//             for c = 0 to N-1
//                 D[r,c] = min (D[r,c], D[r,i] + D[i,c])
int i_root = 0;
for (int i = 0; i < n; ++ i)
   {
   double[] d_i = d[i];
```

The variable `i_root` keeps track of which process owns row *i* of the matrix and hence is the root of the broadcast. Whenever *i* goes outside the range of the current root's row slice, the root advances to the next process.

```
   // Determine which process owns row i.
   if (! ranges[i_root].contains (i)) ++ i_root;

   // Broadcast row i from owner process to all processes.
   if (rank == i_root)
      {
      world.broadcast (i_root, DoubleBuf.buffer (d_i));
      }
   else
      {
      world.broadcast (i_root, row_i_buf);
      d_i = row_i;
      }

   // Inner loops over rows in my slice and over all columns.
   for (int r = mylb; r <= myub; ++ r)
      {
      double[] d_r = d[r];
      for (int c = 0; c < n; ++ c)
         {
         d_r[c] = Math.min (d_r[c], d_r[i] + d_i[c]);
         }
      }
   }

long t3 = System.currentTimeMillis();

// Write distance matrix slice to a separate output file in
// each process.
DoubleMatrixFile out = new DoubleMatrixFile (n, n, d);
DoubleMatrixFile.Writer writer =
   out.prepareToWrite
      (new BufferedOutputStream
         (new FileOutputStream
            (Files.fileForRank (outfile, rank))));
writer.writeRowSlice (myrange);
writer.close();

// Stop timing.
long t4 = System.currentTimeMillis();
System.out.println ((t2-t1) + " msec pre " + rank);
System.out.println ((t3-t2) + " msec calc " + rank);
```

```
         System.out.println ((t4-t3) + " msec post " + rank);
         System.out.println ((t4-t1) + " msec total " + rank);
         }
      }
```

## 25.4 Message Broadcast Time Model

Before looking at the running-time measurements for the FloydClu program, let's derive a model to predict the running time for the computation portion (omitting the file I/O). To do that, we need to derive a model to predict the time needed to broadcast a message, analogous to the message send time model in Chapter 24. To do that, we need to understand how the Parallel Java Library implements message broadcasting.

Suppose there are $K$=8 processes in the parallel program and process 0 is the root of the broadcast. The naive way to broadcast would be for process 0 to send a copy of the message to each process in turn (Figure 25.4). Done this way, broadcasting a message takes ($K$–1) times as long as sending a message to one process.

But broadcasting this way fails to take advantage of the cluster's ability to do several things at once in separate processors. The Parallel Java Library accomplishes the broadcast in less time by sending messages in parallel (Figure 25.5). First, process 0 sends the message to process 1. Because process 1 now has a copy of the message, process 1 can send the message to process 3 while at the same time process 0 sends the message to process 2. Now processes 0–3 all have a copy of the message, and to finish the broadcast, processes 0–3 simultaneously send the message to processes 4–7, respectively.

Done this way, broadcasting a message takes ($\log_2 K$) times as long as sending a message to one process. In other words, a broadcast takes $O(\log K)$ time instead of $O(K)$ time. A broadcast remains efficient even as the program scales up to many processes. However, note that in the final round, half the processes are sending messages to the other half at the same time. To avoid diminishing the program's performance, the cluster backend network must have a high bisection bandwidth, as stated in Chapter 2.

Figure 25.6 shows the pattern of messages sent from process to process during a broadcast. Because the graph of communicating processes is a tree, this pattern is called a **broadcast tree**. The process at the root of the tree, naturally, is the root of the broadcast. The message pattern can be drawn to emphasize the levels in the broadcast tree (the first view in Figure 25.6) or to depict the processes occupying the corners of a hypercube (the second view). In the latter view, each round of messages is sent along a different dimension of the hypercube.

Now we can derive a formula for the time to do a broadcast: it is the time to send a message multiplied by the number of levels in the broadcast tree. The former quantity is given by the message send-time model, such as Equation 24.1 for the "tardis" computer. The number of levels is ceil($\log_2 K$), where ceil is the ceiling function (the smallest integer greater than or equal to the argument). Putting the two together gives the **message broadcast time model** for the "tardis" computer,

$$T(N) = (2.08 \times 10^{-4} + 1.07 \times 10^{-9} N) \cdot \mathrm{ceil}(\log_2 K) \tag{25.1}$$

where *T* is the broadcast time in seconds, *N* is the message size in bits, and *K* is the number of processes. This formula assumes an ideal bisection bandwidth, so that no matter how many messages are sent during a round, the round takes the same time as sending one message.

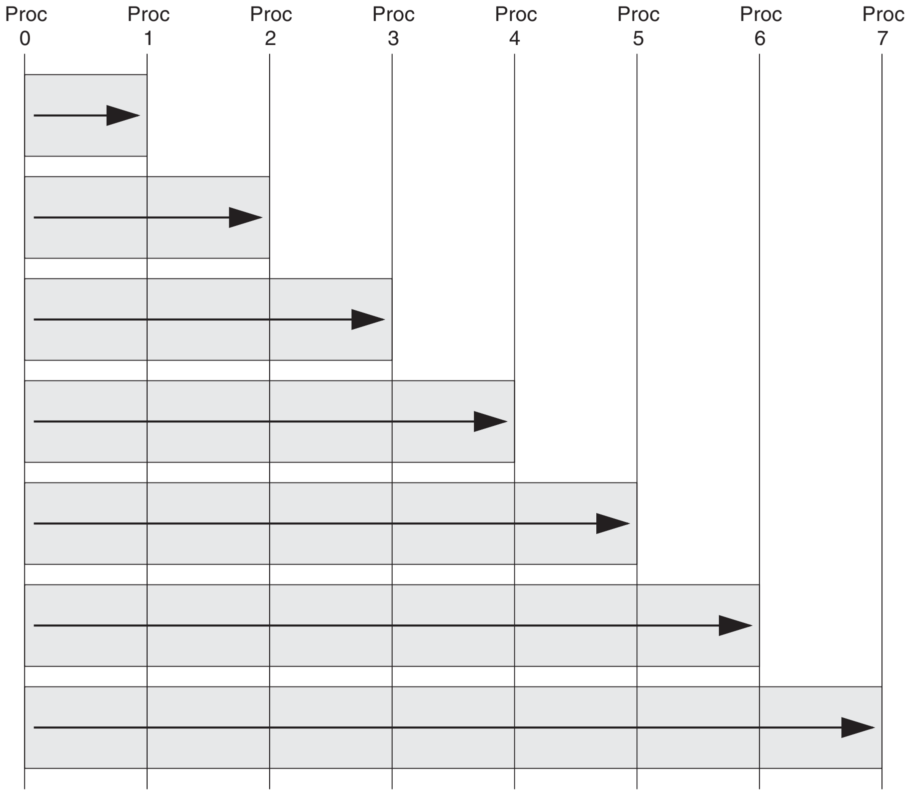


**Figure 25.4** Broadcast by sending one message at a time

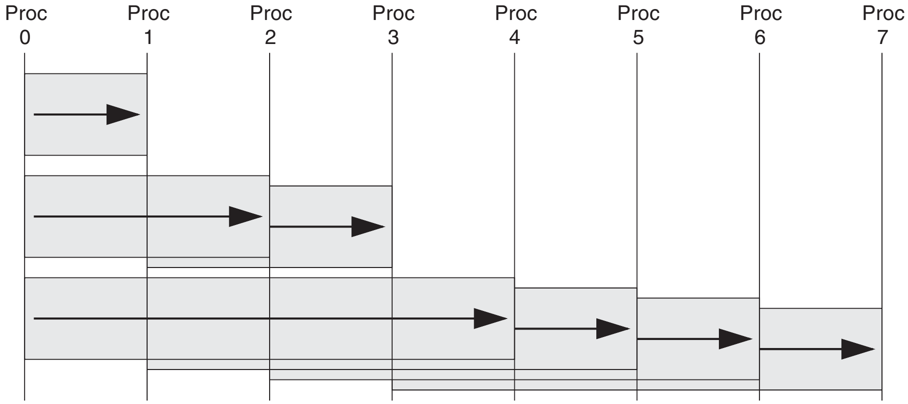


**Figure 25.5** Broadcast by sending multiple messages in parallel

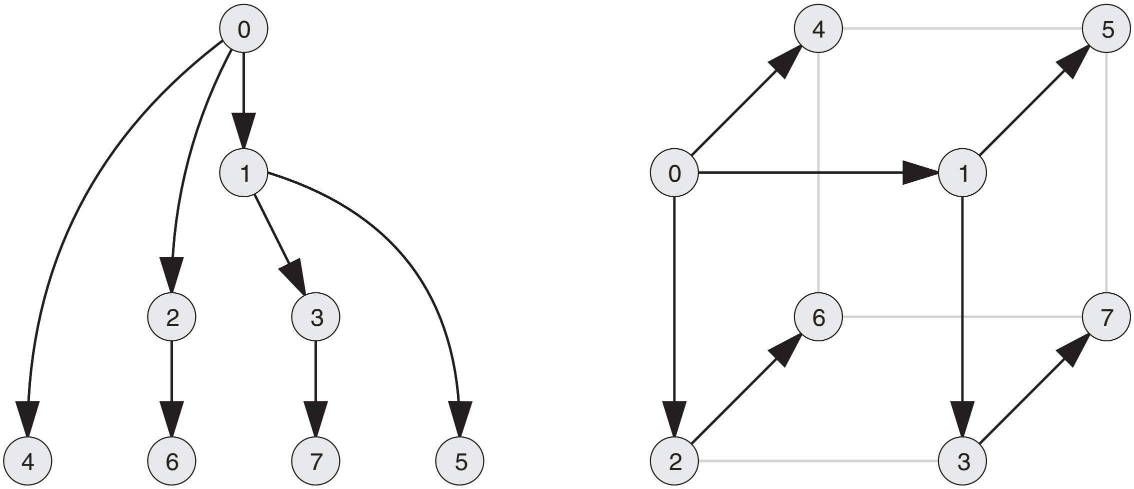


**Figure 25.6** Broadcast message pattern for $K = 8$ processes with root process 0, drawn as a three-level tree and as a three-dimensional hypercube

## 25.5 Computation Time Model

Now we can derive a model for the running time of the computation portion of the cluster parallel Floyd's Algorithm program. The running time consists of the time to calculate the distance matrix plus the time to do all the broadcasts.

We can determine the time to calculate the distance matrix by measuring the sequential version's running time. For an $n \times n$ distance matrix, the sequential version's running time is proportional to $n^3$; this includes the loop control statements and the assignment statements in the inner loop. We will assume that the running times for the loop control statements in the middle and outer loops, which are proportional to $n^2$ and $n$ respectively, are negligible compared to the inner loop's running time. Table 25.1 gives the running-time measurements for the sequential version on the "tardis" computer. If we fit the data to the power function model $T_{\text{calc}}(n,1) = a \cdot n^3$, we find the coefficient $a = 8.86 \times 10^{-9}$ gives the best fit. (See Appendix C for further information about fitting to a power function model.) Table 25.1 also shows the $T_{\text{calc}}(n,1)$ values computed by the model.

**Table 25.1** Sequential version running times

| $n$ | Measured $T_{\text{calc}}(n,1)$ (sec) | Model $T_{\text{calc}}(n,1)$ (sec) |
|---:|---:|---:|
| 2000 | 67.942 | 70.889 |
| 2520 | 136.506 | 141.805 |
| 3180 | 269.528 | 284.952 |
| 4000 | 589.131 | 567.115 |
| 5040 | 1182.404 | 1134.443 |
| 6360 | 2474.565 | 2279.619 |

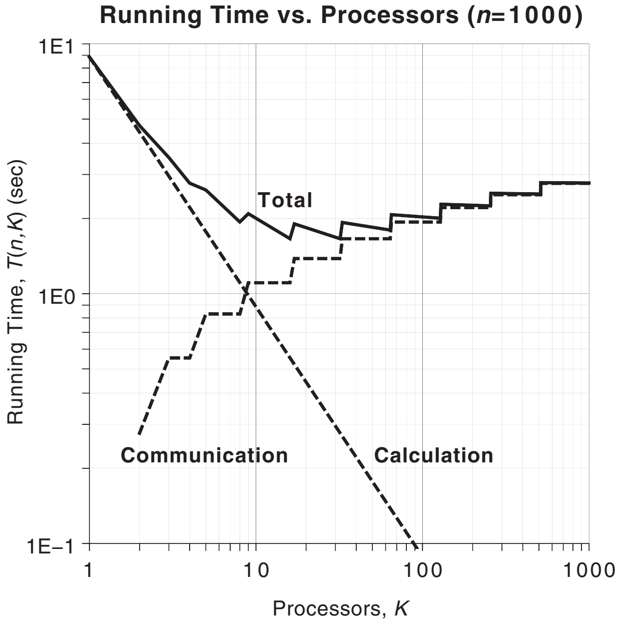


**Figure 25.7** T(*n*,*K*) predicted by computation time model

When we run the parallel program on *K* processors, the calculation speeds up by a factor of *K*. The parallel program's calculation time therefore is the following:

$$T_{\text{calc}}(n,K) = 8.86\times10^{-9}n^3 \,/\, K \tag{25.2}$$

For an *n*×*n* distance matrix, the parallel version does *n* broadcasts. Each broadcast consists of *n* double values, or 64*n* bits. Plugging this into Equation 25.1, the parallel program's broadcast time is the following:

$$T_{\text{bcast}}(n,K) = n(2.08\times10^{-4} + 6.85\times10^{-8}n)\cdot\ \text{ceil}(\log_2 K) \tag{25.3}$$

Adding Equations 25.2 and 25.3 together gives the computation time model for the parallel program,

$$\begin{aligned} T(n,K) \;&=\; 8.86\times10^{-9}n^3 \,/\, K \\ &\quad + n(2.08\times10^{-4} + 6.85\times10^{-8}n)\cdot\text{ceil}(\log_2 K) \end{aligned} \tag{25.4}$$

where *n* is the number of vertices in the network (the number of rows and columns in the distance matrix), *K* is the number of processors, and *T* is the running time in seconds.

Consider what the computation time model tells us about the program's performance as the number of processors *K* scales up. The model has one term proportional to (1/*K*) and one term proportional to (log $K$); that is, the model has one term that diminishes and one term that increases as $K$ increases, as plotted in Figure 25.7 for $n$ = 1,000. (The second term goes up by steps because of the ceiling function.) For small $K$, the first term dominates, and the running time decreases as $K$ increases. But at some point the second term becomes larger than the first term. After that, the running time starts going up again.

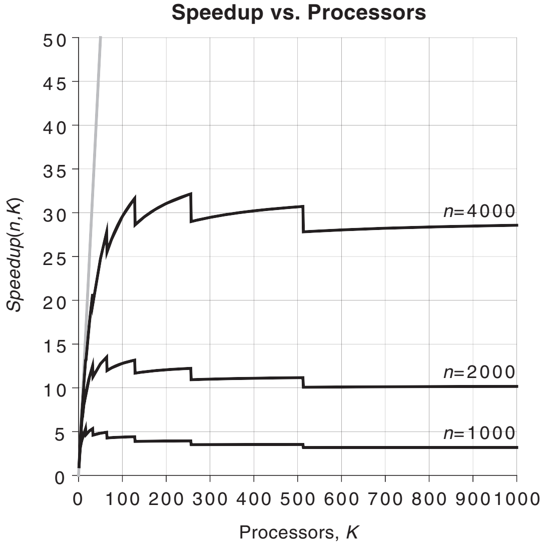


**Figure 25.8** *Speedup*(*n*,*K*) predicted by computation time model

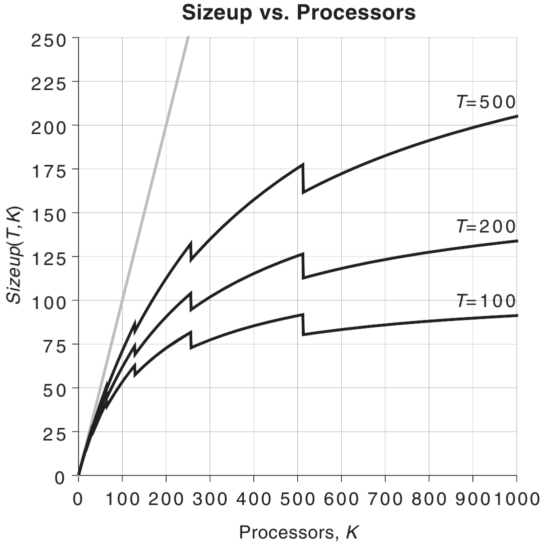


**Figure 25.9** *Sizeup*(*T*,*K*) predicted by computation time model

The speedup predicted by the computation time model is the following:

$$Speedup(n,K) = \frac{T(n,1)}{T(n,K)} \tag{25.5}$$

As shown in Figure 25.8, the speedup starts out increasing as $K$ increases, but at some point the speedup starts going back down. (The vertical scale is exaggerated to make the speedup reduction visible.) Unlike Amdahl's Law, the computation time model for the Floyd's Algorithm cluster parallel program predicts that the speedup can achieve only a certain maximum value, after which adding more processors causes a *slowdown* rather than a further speedup. The slowdown, of course, is due to the always-increasing communication time. Not only does message passing in a cluster parallel program limit the speedup by increasing the sequential fraction, message passing can sometimes turn a speedup into a slowdown. This is another reason to design cluster parallel programs to use the fewest possible messages.

What is the maximum speedup the program can achieve? To find out, we need the value of $K$ that results in the minimum running time. To find that value, we differentiate $T(n,K)$ with respect to $K$, set the derivative equal to 0, and solve for $K$. To simplify finding the derivative, we approximate ceil($\log_2 K$) as just ($\log_2 K$), which is (log $K$/log 2). Then the derivative is the following:

$$\frac{\partial T(n,K)}{\partial K} = -\frac{8.86\times 10^{-9} n^3}{K^2} + \frac{n(2.08\times 10^{-4} + 6.85\times 10^{-8} n)}{K \log 2} \tag{25.6}$$

Set the derivative equal to 0 and multiply both sides by $K^2$:

$$-8.86 \times 10^{-9} n^3 + \frac{n(2.08 \times 10^{-4} + 6.85 \times 10^{-8} n)}{\log 2} K = 0 \tag{25.7}$$

Now solve for $K$:

$$K_{\text{best}} = \frac{6.14 \times 10^{-9} n^2}{2.08 \times 10^{-4} + 6.85 \times^{-8} n} \tag{25.8}$$

For a certain number of vertices $n$, Equation 25.8 gives an estimate for the number of processors $K$ that yields the smallest running time, hence the largest speedup. We refer to this value as $K_{\text{best}}$. For $n = 1{,}000$ vertices, the maximum speedup comes at $K_{\text{best}} \approx 22$. We can get the actual maximum speedup value by plugging $K_{\text{best}}$ into Equation 25.5; for example, $\mathit{Speedup}(1{,}000, 22) = 4.963$. Notice what Equation 25.8 says about the program's performance as the number of vertices $n$ increases. Because $K_{\text{best}}$ is $O(n)$, as the number of vertices increases we can scale up to more processors before hitting the slowdown.

What about sizeup? The problem size $N$ is the number of calculations the program has to do, $N = n^3$. The sizeup is the following:

$$\mathit{Sizeup}(T, K) = \frac{N(T, K)}{N(T, 1)} = \left[ \frac{n(T, K)}{n(T, 1)} \right]^3 \tag{25.9}$$

To compute the sizeup, we must first compute the number of vertices $n$ in the graph that can be solved in time $T$ on $K$ processors, $n(T,K)$. For a given $T$ and $K$, Equation 25.4 becomes a cubic equation in $n$, and solving the cubic equation yields the value of $n$. (See Appendix C for how to solve a cubic equation.) Figure 25.9 plots the sizeups predicted for the cluster parallel Floyd's Algorithm program. While less than ideal, the sizeups scale up much better than the speedups as $K$ increases. Consider a 4,000-vertex graph, which takes about 500 seconds to run on one processor. On 1,000 processors, the sizeup for this running time is about 200, but the speedup for this problem size is only about 30. Once again, we see that going for sizeup is preferable to going for speedup.

## 25.6 Parallel Floyd's Algorithm Performance

Table 25.2 (at the end of the chapter) lists, and Figure 25.10 plots, the FloydClu program's performance on the "tardis" parallel computer. The running times are for the calculation portion only, and do not include the distance matrix file I/O. The program was run on distance matrices with the following listed numbers of vertices $n$ and problem sizes $N = n^3$.

| $n$ | $N$ | |
|---|---|---|
| 2,000 | 8,000,000,000 | (8G) |
| 2,520 | 16,003,008,000 | (16G) |
| 3,180 | 32,157,432,000 | (32G) |
| 4,000 | 64,000,000,000 | (64G) |
| 5,040 | 128,024,064,000 | (128G) |
| 6,360 | 257,259,456,000 | (256G) |

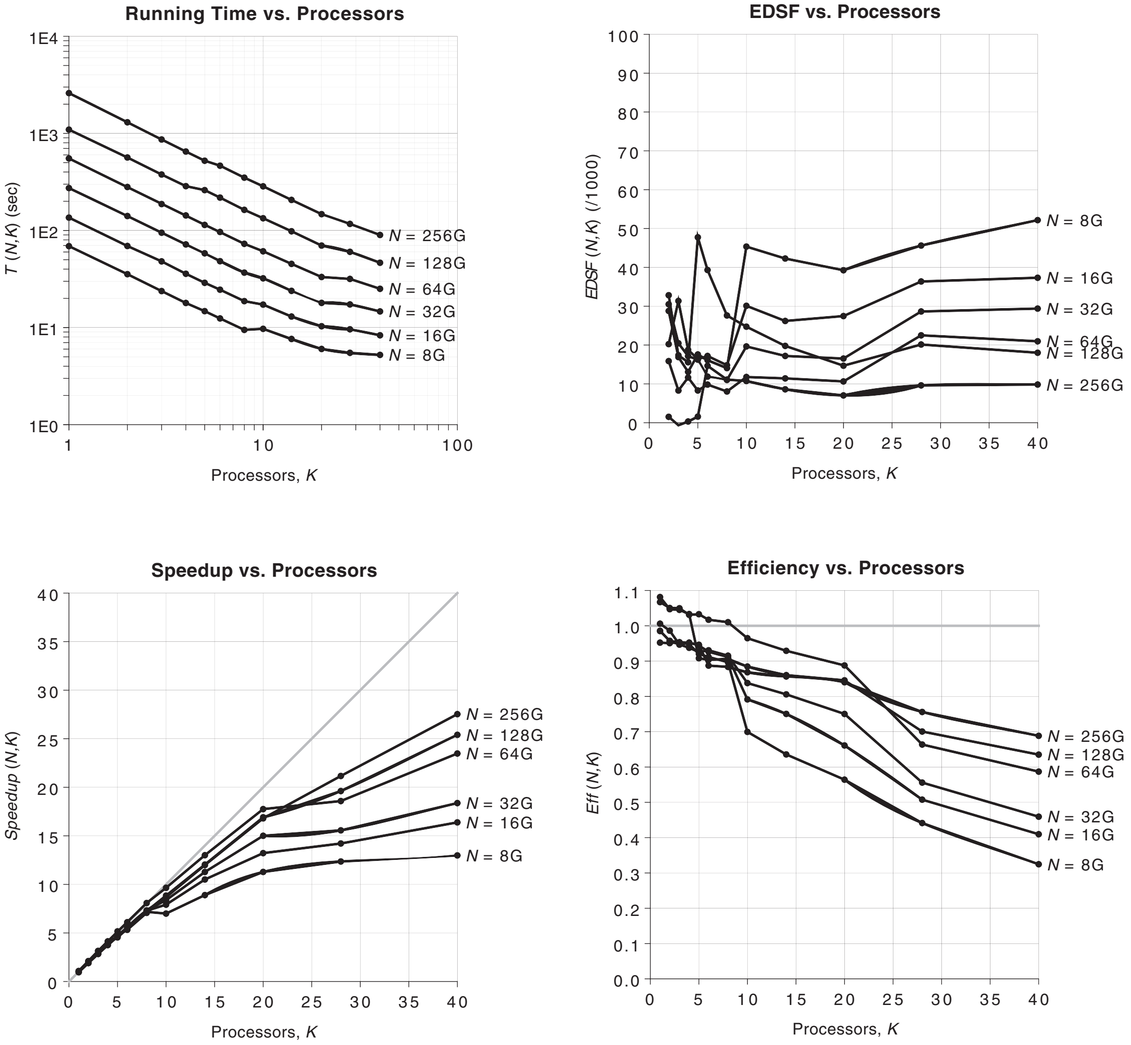


**Figure 25.10** FloydSeq/FloydClu running-time metrics

Each distance matrix input file was created with the following command, where `$n` stands for one of the preceding *n* values:

```
$ java edu.rit.smp.network.FloydRandom 142857 0.25 $n in$n.dat
```

The effect of the message broadcasts on the speedups, efficiencies, and *EDSFs* is clearly visible in the plots. The program scales up well until about 10 processors, and then the metrics rapidly get worse. We don't see the slowdown predicted by the computation time model because the "tardis" computer lacks enough processors to scale up to the point where the slowdown starts. (For $n$ = 2,000, Equation 25.8 says $K_{\text{best}} \approx 71$, but the "tardis" computer only has 40 processors.)

How well does the computation time model predict the FloydClu program's running time? Equation 25.4 assumes all messages are inter-node messages. Thus, Equation 25.4 is applicable when each process runs on a different node, as is the case for $K$=10 and below. But for $K$=14 and 20, each node runs two processes (Figure 25.11). In this case, the first round of the broadcast is an intra-node message, and the other rounds are inter-node. The broadcast time is therefore the time to send one intra-node message plus the time to send ceil($\log_2 K$)–1 inter-node messages:

$$\begin{aligned} T(n,K) = {} & 8.86\times10^{-9} n^3 / K \\ & + n(2.08\times10^{-4} + 6.85\times10^{-8} n) \\ & + n(7.89\times10^{-5} + 1.45\times10^{-8} n)\cdot(\text{ceil}(\log_2 K) - 1) \end{aligned} \tag{25.10}$$

For $K$=28 and 40, each node runs four processes. In this case, the first two message rounds are intra-node, and the remaining rounds are inter-node. The broadcast time is therefore the time to send two intra-node messages plus the time to send ceil($\log_2 K$)–2 inter-node messages:

$$\begin{aligned} T(n,K) = {} & 8.86\times10^{-9} n^3 / K \\ & + n(2.08\times10^{-4} + 6.85\times10^{-8} n)\cdot 2 \\ & + n(7.89\times10^{-5} + 1.45\times10^{-8} n)\cdot(\text{ceil}(\log_2 K) - 2) \end{aligned} \tag{25.11}$$

Figure 25.12 plots the measured running times, and the running times predicted by Equation 25.4, 25.10, or 25.11, as appropriate, for a graph with $n$ = 2,000 vertices ($N$ = 8G). The predicted time is within 8 percent of the actual time for the majority of the data points and is within 15 percent for all the data points.

To sum up, the Floyd's Algorithm program's running time measurements reinforce what we said in Chapter 24: To get good parallel performance, *the program must have much more computation than communication.* Unfortunately, the cluster parallel version of Floyd's Algorithm doesn't have enough computation relative to communication, at least not for the problem sizes we measured. A modern CPU can execute the statement "$d_{rc} \leftarrow \min(d_{rc}, d_{ri} + d_{ic})$" in almost no time at all; broadcasting a message takes forever, comparatively speaking. Floyd's Algorithm is just not well suited to run on a cluster parallel computer.

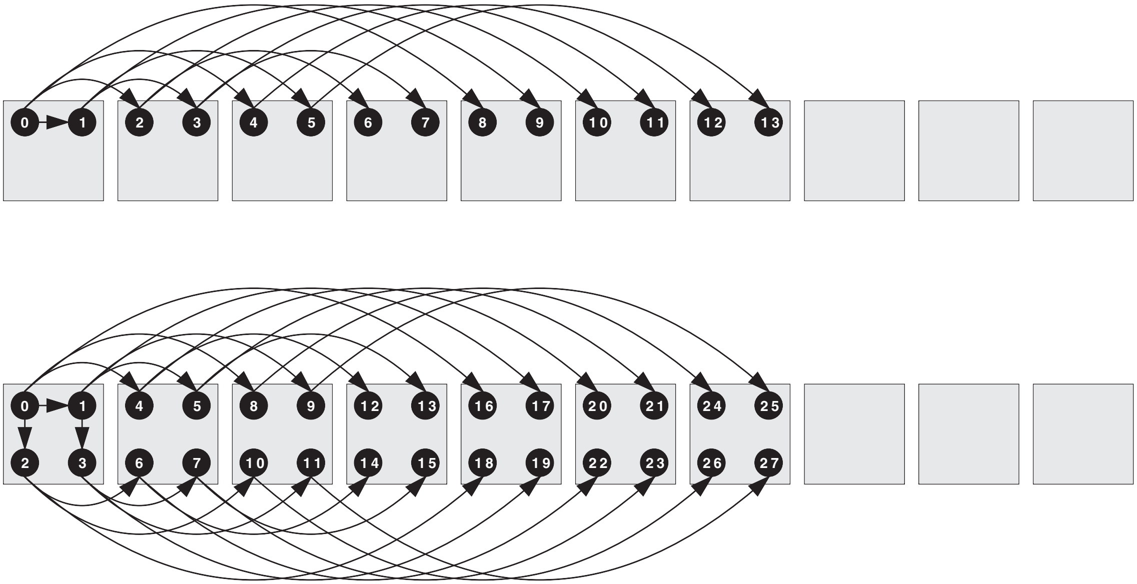


**Figure 25.11** Floyd's Algorithm program process allocation and message broadcasting on the "tardis" computer

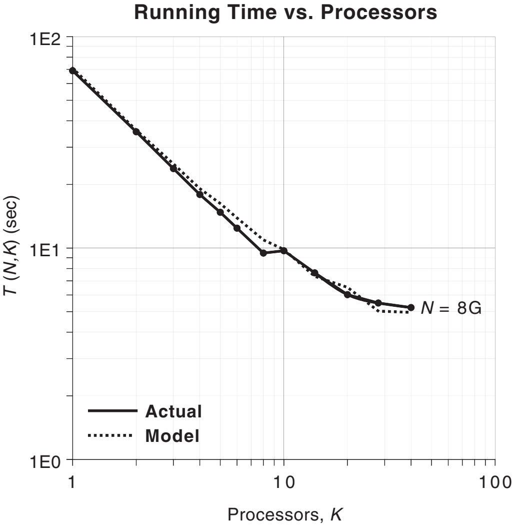
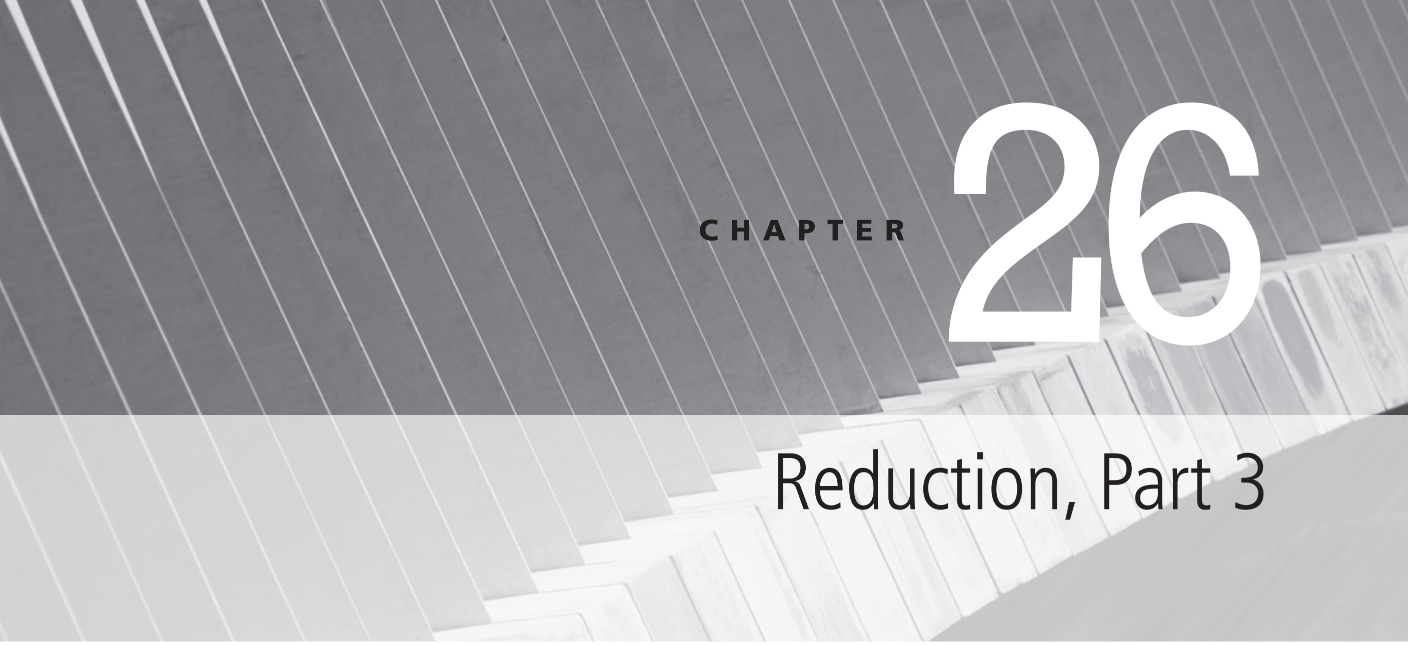


**Figure 25.12** FloydClu predicted and actual running times

**Table 25.2** FloydSeq/FloydClu running-time metrics

| *N* | *K* | *T* | *Spdup* | *Eff* | *EDSF* | *N* | *K* | *T* | *Spdup* | *Eff* | *EDSF* |
|---:|---:|---:|---:|---:|---:|---:|---:|---:|---:|---:|---:|
| 8G | seq | 67942 | | | | 64G | seq | 589131 | | | |
| 8G | 1 | 68971 | 0.985 | 0.985 | | 64G | 1 | 552104 | 1.067 | 1.067 | |
| 8G | 2 | 35478 | 1.915 | 0.958 | 0.029 | 64G | 2 | 280431 | 2.101 | 1.050 | 0.016 |
| 8G | 3 | 23767 | 2.859 | 0.953 | 0.017 | 64G | 3 | 187085 | 3.149 | 1.050 | 0.008 |
| 8G | 4 | 17920 | 3.791 | 0.948 | 0.013 | 64G | 4 | 142842 | 4.124 | 1.031 | 0.012 |
| 8G | 5 | 14764 | 4.602 | 0.920 | 0.018 | 64G | 5 | 114086 | 5.164 | 1.033 | 0.008 |
| 8G | 6 | 12421 | 5.470 | 0.912 | 0.016 | 64G | 6 | 96536 | 6.103 | 1.017 | 0.010 |
| 8G | 8 | 9470 | 7.174 | 0.897 | 0.014 | 64G | 8 | 72898 | 8.082 | 1.010 | 0.008 |
| 8G | 10 | 9712 | 6.996 | 0.700 | 0.045 | 64G | 10 | 61058 | 9.649 | 0.965 | 0.012 |
| 8G | 14 | 7634 | 8.900 | 0.636 | 0.042 | 64G | 14 | 45288 | 13.009 | 0.929 | 0.011 |
| 8G | 20 | 6021 | 11.284 | 0.564 | 0.039 | 64G | 20 | 33180 | 17.756 | 0.888 | 0.011 |
| 8G | 28 | 5497 | 12.360 | 0.441 | 0.046 | 64G | 28 | 31699 | 18.585 | 0.664 | 0.023 |
| 8G | 40 | 5232 | 12.986 | 0.325 | 0.052 | 64G | 40 | 25090 | 23.481 | 0.587 | 0.021 |
| 16G | seq | 136506 | | | | 128G | seq | 1182404 | | | |
| 16G | 1 | 135688 | 1.006 | 1.006 | | 128G | 1 | 1093355 | 1.081 | 1.081 | |
| 16G | 2 | 69217 | 1.972 | 0.986 | 0.020 | 128G | 2 | 564624 | 2.094 | 1.047 | 0.033 |
| 16G | 3 | 48065 | 2.840 | 0.947 | 0.031 | 128G | 3 | 377108 | 3.135 | 1.045 | 0.017 |
| 16G | 4 | 35832 | 3.810 | 0.952 | 0.019 | 128G | 4 | 286107 | 4.133 | 1.033 | 0.016 |
| 16G | 5 | 28916 | 4.721 | 0.944 | 0.016 | 128G | 5 | 260432 | 4.540 | 0.908 | 0.048 |
| 16G | 6 | 24558 | 5.559 | 0.926 | 0.017 | 128G | 6 | 218056 | 5.422 | 0.904 | 0.039 |
| 16G | 8 | 18719 | 7.292 | 0.912 | 0.015 | 128G | 8 | 163085 | 7.250 | 0.906 | 0.028 |
| 16G | 10 | 17245 | 7.916 | 0.792 | 0.030 | 128G | 10 | 133669 | 8.846 | 0.885 | 0.025 |
| 16G | 14 | 12993 | 10.506 | 0.750 | 0.026 | 128G | 14 | 98180 | 12.043 | 0.860 | 0.020 |
| 16G | 20 | 10324 | 13.222 | 0.661 | 0.027 | 128G | 20 | 69919 | 16.911 | 0.846 | 0.015 |
| 16G | 28 | 9604 | 14.213 | 0.508 | 0.036 | 128G | 28 | 60279 | 19.616 | 0.701 | 0.020 |
| 16G | 40 | 8330 | 16.387 | 0.410 | 0.037 | 128G | 40 | 46541 | 25.406 | 0.635 | 0.018 |
| 32G | seq | 269528 | | | | 256G | seq | 2474565 | | | |
| 32G | 1 | 273386 | 0.986 | 0.986 | | 256G | 1 | 2598537 | 0.952 | 0.952 | |
| 32G | 2 | 140863 | 1.913 | 0.957 | 0.031 | 256G | 2 | 1301244 | 1.902 | 0.951 | 0.002 |
| 32G | 3 | 94860 | 2.841 | 0.947 | 0.020 | 256G | 3 | 865066 | 2.861 | 0.954 | -0.001 |
| 32G | 4 | 71861 | 3.751 | 0.938 | 0.017 | 256G | 4 | 650250 | 3.806 | 0.951 | 0.000 |
| 32G | 5 | 58229 | 4.629 | 0.926 | 0.016 | 256G | 5 | 522983 | 4.732 | 0.946 | 0.002 |
| 32G | 6 | 48270 | 5.584 | 0.931 | 0.012 | 256G | 6 | 464837 | 5.324 | 0.887 | 0.015 |
| 32G | 8 | 36806 | 7.323 | 0.915 | 0.011 | 256G | 8 | 350066 | 7.069 | 0.884 | 0.011 |
| 32G | 10 | 32171 | 8.378 | 0.838 | 0.020 | 256G | 10 | 284977 | 8.683 | 0.868 | 0.011 |
| 32G | 14 | 23892 | 11.281 | 0.806 | 0.017 | 256G | 14 | 206371 | 11.991 | 0.856 | 0.009 |
| 32G | 20 | 17960 | 15.007 | 0.750 | 0.017 | 256G | 20 | 147288 | 16.801 | 0.840 | 0.007 |
| 32G | 28 | 17314 | 15.567 | 0.556 | 0.029 | 256G | 28 | 116903 | 21.168 | 0.756 | 0.010 |
| 32G | 40 | 14668 | 18.375 | 0.459 | 0.029 | 256G | 40 | 89872 | 27.534 | 0.688 | 0.010 |



# CHAPTER 26

# Reduction, Part 3

in which we learn how to implement the reduction pattern in a cluster parallel program; we learn how to reduce single variables and arrays; and we see how reduction is performed efficiently on a cluster

# CHAPTER 26 Reduction, Part 3

## 26.1 Estimating pi on a Cluster

In Chapter 13, we used this Monte Carlo algorithm to compute an estimate for π (Figure 26.1):

```
$C \leftarrow 0$
Repeat $N$ times:
    $x \leftarrow$ random (0, 1)
    $y \leftarrow$ random (0, 1)
    If $x^2 + y^2 \leq 1$: $C \leftarrow C + 1$
Print $4C/N$ as the estimate for $\pi$
```

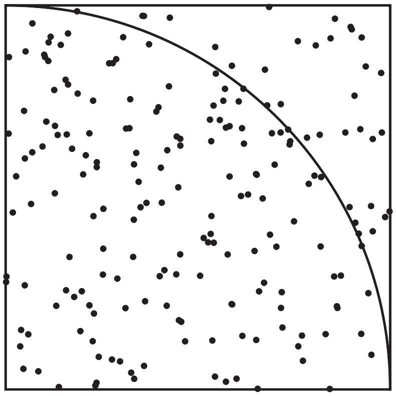


**Figure 26.1** A dartboard for estimating π

To run this algorithm on a cluster parallel computer, we will do as we did on the SMP parallel computer. We will partition the $N$ iterations among the $K$ processes, each process doing $N/K$ iterations. Also, each process will update its own local counter. After all the iterations, the processes' local counters will be *reduced* into one counter using addition as the reduction operator. That counter will then be used to calculate π.

In the SMP parallel version, we had a choice between two designs. In one, all the threads updated a global shared counter variable directly. In the other, each thread updated its own local counter variable, and then the local counters were reduced into a shared global counter. We chose the second alternative because it gave better performance.

In the cluster parallel version, we have a similar choice. We could choose to have one global counter variable in one of the processes, process 0, say. But because variables can't be shared between processes on a cluster, the other processes would have to send a message to process 0 each time they needed to increment the counter. All this message passing would kill the cluster program's performance—just as all the thread synchronization on the global shared variable decreased the SMP program's performance. Instead, we will go with the *reduction* pattern for the cluster program. That way, only one round of message passing has to happen, and the program's performance should be decreased only a little.

Likewise, each process will have its own local pseudorandom number generator (PRNG) rather than using a shared global PRNG. To ensure that the cluster parallel version computes the same answer as the sequential version, we will use the *sequence splitting* technique. Each process will initialize its PRNG with the same seed, and then skip its PRNG ahead the proper amount.

## 26.2 Collective Communication: Reduction

Before looking at the code for the cluster parallel π program, let's consider how a cluster parallel program sends messages among the processes to do a reduction.

Reduction is the opposite of broadcast. Instead of disseminating data from one process to all $K$ processes, reduction concentrates data from all $K$ processes into one process. Consequently, like a broadcast, a reduction can be done with ceil($\log_2 K$) rounds of messages (Figure 26.2). For a reduction with $K = 8$ processes into root process 0, processes 4–7 first send messages to processes 0–3; then processes 2 and 3 send messages to processes 0 and 1; and, finally, process 1 sends a message to process 0. As each process receives a message, it combines the data in the message with the data in the process's own buffer using the reduction operator.

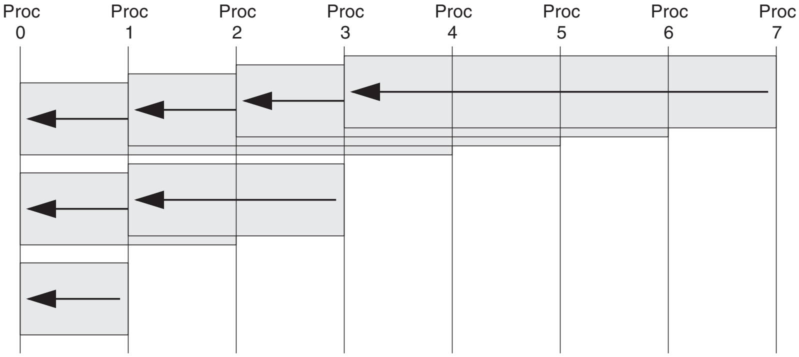


**Figure 26.2** Reduction by sending multiple messages in parallel

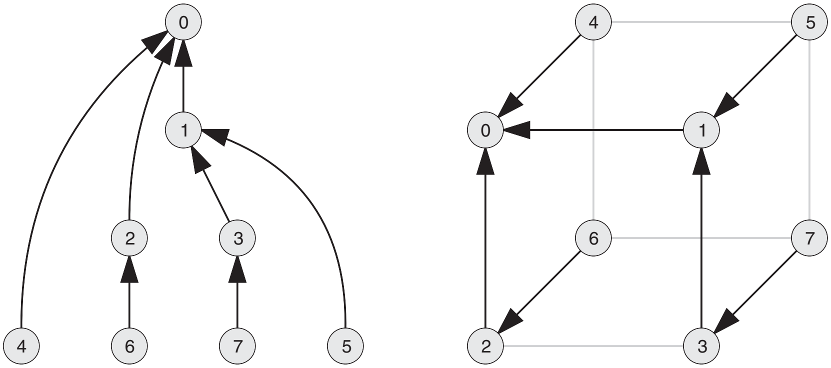


**Figure 26.3** Reduction message pattern for $K = 8$ processes with root process 0, drawn as a three-level tree and as a three-dimensional hypercube

Figure 26.3 shows the pattern of messages sent from process to process during a reduction—the **reduction tree**. The process at the root of the tree is the root of the reduction. Figure 26.4 shows what happens to the data buffers during a reduction operation among $K = 8$ processes into root process 0 using addition as the reduction operator. Each buffer has an initial value. During the first round of message passing, process 4 sends its buffer value, 46, to process 0; process 0 feeds its buffer value, 56, along with the 46 from process 4 into the reduction operator; process 0 stores the result, 102, back into its buffer. The same happens in parallel between processes 5 and 1, 6 and 2, and 7 and 3. In the second round of message passing, process 2's buffer (which holds a reduction result from the first round) is reduced into process 0's buffer; in parallel, process 3's buffer is reduced into process 1's buffer. In the third and final message round, process 1's buffer is reduced into process 0's buffer, which ends up holding the sum of all the initial buffer values. The other processes' buffers end up holding intermediate values in the reduction.

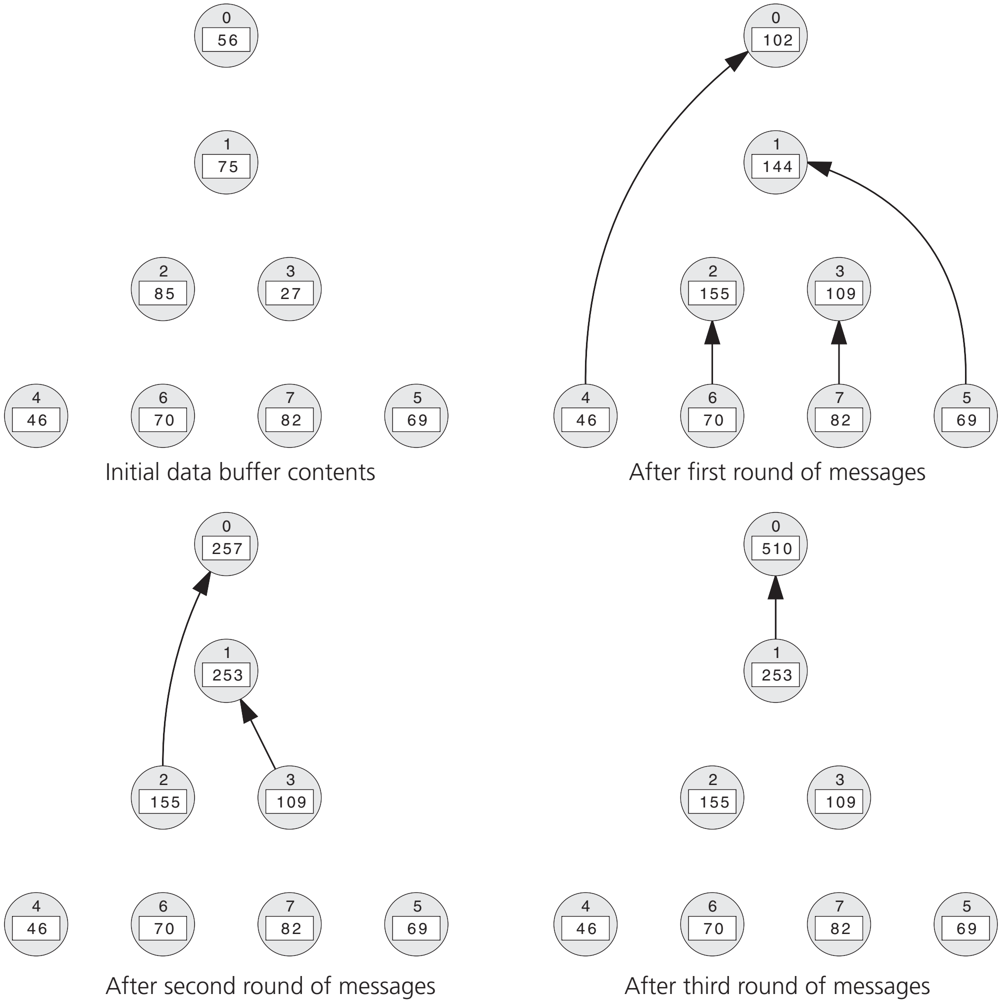

0
56
1
75
2
85
3
27
4
46
6
70
7
82
5
69
Initial data buffer contents
0
102
1
144
2
155
3
109
4
46
6
70
7
82
5
69
After first round of messages
0
257
1
253
2
155
3
109
4
46
6
70
7
82
5
69
After second round of messages
0
510
1
253
2
155
3
109
4
46
6
70
7
82
5
69
After third round of messages

**Figure 26.4** Contents of the data buffers during a reduction operation with addition as the reduction operator

You don't need to code all this message passing yourself. All you have to code is this statement in each process.

```
    world.reduce (0, buf, IntegerOp.SUM);
```

The Parallel Java message passing layer then sends and receives all the messages and invokes the reduction operator. However, it's important to understand what's going on under the hood in the `reduce()` method.

## 26.3 Parallel pi Program with Reduction

Here is the code for the cluster parallel version of the Monte Carlo π program, class edu.rit.clu.monte.PiClu.

```
package edu.rit.clu.monte;
import edu.rit.mp.buf.LongItemBuf;
import edu.rit.pj.Comm;
import edu.rit.pj.reduction.LongOp;
import edu.rit.util.LongRange;
import edu.rit.util.Random;
public class PiClu
   {
   // World communicator.
   static Comm world;
   static int size;
   static int rank;

   // Command line arguments.
   static long seed;
   static long N;

   // Pseudorandom number generator.
   static Random prng;

   // Number of points within the unit circle.
   static long count;

   /**
    * Main program.
    */
   public static void main
      (String[] args)
      throws Exception
      {
      // Start timing.
      long time = -System.currentTimeMillis();

      // Initialize middleware.
      Comm.init (args);
      world = Comm.world();
      size = world.size();
      rank = world.rank();

      // Validate command line arguments.
      if (args.length != 2) usage();
      seed = Long.parseLong (args[0]);
      N = Long.parseLong (args[1]);

      // Determine range of iterations for this process.
      LongRange range =
         new LongRange (0, N-1) .subrange (size, rank);
      long my_N = range.length();

      // Set up PRNG and skip ahead over the random numbers the
      // lower-ranked processes will generate.
      prng = Random.getInstance (seed);
      prng.skip (2 * range.lb());

      // Generate random points in the unit square, count how many
      // are in the unit circle.
      count = 0L;
      for (long i = 0L; i < my_N; ++ i)
         {
         double x = prng.nextDouble();
         double y = prng.nextDouble();
         if (x*x + y*y <= 1.0) ++ count;
         }
```

Here is the key statement in the parallel program: the reduction into root process 0. After the reduction, only process 0 prints the answer.

```
   // Reduce all processes' counts together into process 0.
   LongItemBuf buf = new LongItemBuf();
   buf.item = count;
   world.reduce (0, buf, LongOp.SUM);
   count = buf.item;

   // Stop timing.
   time += System.currentTimeMillis();

   // Print results.
   System.out.println (time + " msec total " + rank);
   if (rank == 0)
      {
      System.out.println
         ("pi = 4 * " + count + " / " + N + " = " +
          (4.0 * count / N));
      }
   }
}
```

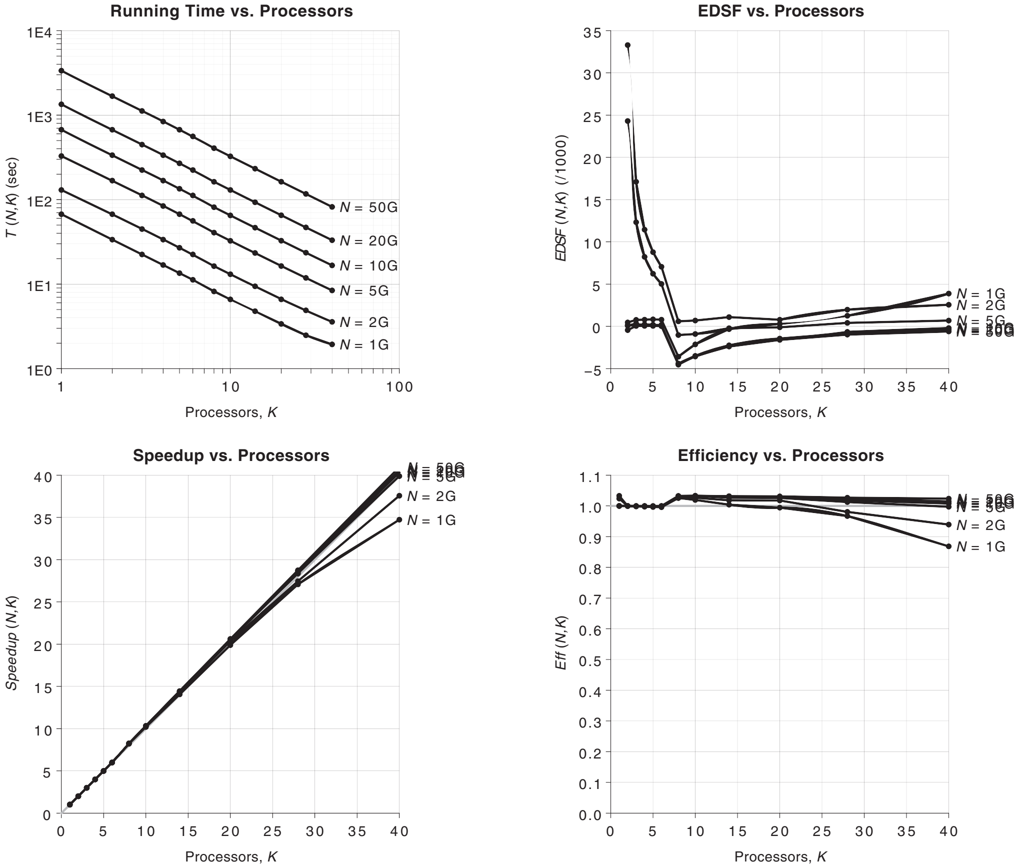


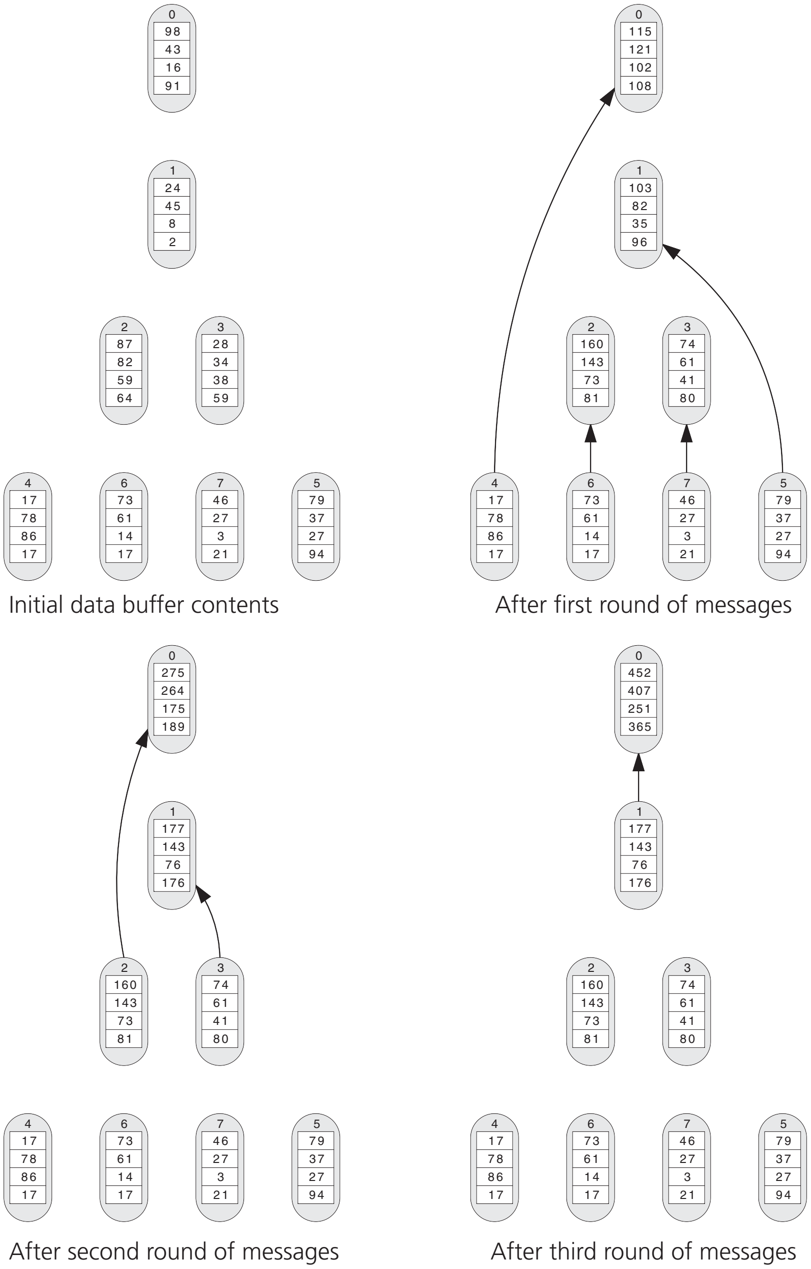
**Figure 26.5** PiSeq/PiClu running-time metrics

Table 26.1 (at the end of the chapter) lists, and Figure 26.5 plots, the PiClu program's performance on the "tardis" parallel computer for problem sizes from 1 billion to 50 billion random points. Due to the small amount of message passing needed for the final reduction, the program achieves good speedups and efficiencies.

## 26.4 Mandelbrot Set Histogram Program

Let's revisit Chapter 15's program to compute a histogram of the Mandelbrot Set and write a cluster parallel version. Recall that the program computed a `histogram` variable, an `int[]` array, where the array index is the pixel value, and the array element is the number of pixels with that value.

Because dividing the pixels equally among the processes results in an unbalanced load, the cluster parallel Mandelbrot Set Histogram program must use the *master-worker* pattern for load balancing. The program also uses the *reduction* pattern, as did the SMP version. Each process updates its own `histogram` array based on the pixel values the process computes. Afterward, the per-process `histogram` arrays are reduced together into process 0 using addition as the reduction operator.



Initial data buffer contents

After first round of messages

After second round of messages

After third round of messages

**Figure 26.6** Contents of the data buffers during an array reduction operation with addition as the reduction operator

To reduce the array variables together into process 0, each process executes these statements.

```
int[] histogram = new int [maxiter + 1];
. . .
world.reduce (0, IntegerBuf.buffer (histogram), IntegerOp.SUM);
```

The reduction buffer encompasses all the elements in the `histogram` array. The `histogram[0]` elements from all the processes are combined using `IntegerOp.SUM` as the reduction operator, and the sum is placed in `histogram[0]` in the root process (process 0). Similarly, the sum of all the `histogram[1]` elements is placed in `histogram[1]` in process 0, and so on for every array index. Figure 26.6 shows the messages sent between processes to accomplish the reduction; in this example, each message contains four integers.

Here is the source code for the cluster parallel version of the Mandelbrot Set histogram program, class edu.rit.clu.fractal.MSHistogramClu.

```
package edu.rit.clu.fractal;
import edu.rit.mp.IntegerBuf;
import edu.rit.mp.ObjectBuf;
import edu.rit.mp.buf.ObjectItemBuf;
import edu.rit.pj.Comm;
import edu.rit.pj.CommStatus;
import edu.rit.pj.IntegerSchedule;
import edu.rit.pj.ParallelRegion;
import edu.rit.pj.ParallelSection;
import edu.rit.pj.ParallelTeam;
import edu.rit.pj.reduction.IntegerOp;
import edu.rit.util.Range;
import java.io.BufferedWriter;
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;
public class MSHistogramClu
   {
   // Communicator.
   static Comm world;
   static int size;
   static int rank;

   // Command line arguments.
   static int width;
   static int height;
   static double xcenter;
   static double ycenter;
   static double resolution;
   static int maxiter;
   static File outfile;

   // Initial pixel offsets from center.
   static int xoffset;
   static int yoffset;

   // Histogram (array of counters indexed by pixel value).
   static int[] histogram;
```

As in the master-worker Mandelbrot Set programs, we use message tags to distinguish different kinds of messages: messages sent from master to worker containing a chunk for the worker to compute, empty messages sent from worker to master saying the worker has finished a chunk, and messages to perform the reduction on the `histogram` array.

```
   // Message tags.
   static final int WORKER_MSG = 0;
   static final int MASTER_MSG = 1;
   static final int HISTOGRAM_DATA_MSG = 2;

   // Number of chunks the worker computed.
   static int chunkCount;

   /**
    * Mandelbrot Set main program.
    */
   public static void main
      (String[] args)
      throws Exception
      {
      // Start timing.
      long t1 = System.currentTimeMillis();

      // Initialize middleware.
      Comm.init (args);
      world = Comm.world();
      size = world.size();
      rank = world.rank();

      // Validate command line arguments.
      if (args.length != 7) usage();
      width = Integer.parseInt (args[0]);
      height = Integer.parseInt (args[1]);
      xcenter = Double.parseDouble (args[2]);
      ycenter = Double.parseDouble (args[3]);
      resolution = Double.parseDouble (args[4]);
      maxiter = Integer.parseInt (args[5]);
      outfile = new File (args[6]);

      // Initial pixel offsets from center.
      xoffset = -(width - 1) / 2;
      yoffset = (height - 1) / 2;

      // Create histogram.
      histogram = new int [maxiter + 1];

      long t2 = System.currentTimeMillis();

      // In master process, run master section and worker section
      // in parallel.
      if (rank == 0)
         {
         new ParallelTeam(2).execute (new ParallelRegion()
            {
            public void run() throws Exception
               {
               execute (new ParallelSection()
                  {
                  public void run() throws Exception
                     {
                     masterSection();
                     }
                  },
               new ParallelSection()
                  {
                  public void run() throws Exception
                     {
                     workerSection();
                     }
                  });
               }
            });
         }

      // In worker process, run only worker section.
      else
         {
         workerSection();
         }
```

Here is the reduction. We tag the reduction messages with `HISTOGRAM_DATA_MSG` so the master will not confuse them with the chunk-finished messages from the workers.

```
      // Reduce histogram into process 0.
      world.reduce
         (0,
          HISTOGRAM_DATA_MSG,
          IntegerBuf.buffer (histogram),
          IntegerOp.SUM);

      long t3 = System.currentTimeMillis();

      // Process 0 prints histogram.
      if (rank == 0)
         {
         PrintWriter out =
            new PrintWriter
               (new BufferedWriter
                  (new FileWriter (outfile)));
         for (int i = 0; i <= maxiter; ++ i)
            {
            out.print (i);
            out.print (‘\t’);
            out.print (histogram[i]);
            out.println();
            }
         out.close();
         }

      // Stop timing.
      long t4 = System.currentTimeMillis();
      System.out.println (chunkCount + " chunks " + rank);
      System.out.println ((t2-t1) + " msec pre " + rank);
      System.out.println ((t3-t2) + " msec calc " + rank);
      System.out.println ((t4-t3) + " msec post " + rank);
      System.out.println ((t4-t1) + " msec total " + rank);
      }

   /**
    * Perform the master section.
    */
   private static void masterSection()
      throws IOException
      {
```

```
      int worker;
      Range range;

      // Set up a schedule object to divide the row range into
      // chunks.
      IntegerSchedule schedule = IntegerSchedule.runtime();
      schedule.start (size, new Range (0, height-1));

      // Send initial chunk range to each worker. If range is null,
      // no more work for that worker. Keep count of active workers.
      int activeWorkers = size;
      for (worker = 0; worker < size; ++ worker)
         {
         range = schedule.next (worker);
         world.send (worker, WORKER_MSG, ObjectBuf.buffer (range));
         if (range == null) -- activeWorkers;
         }

      // Repeat until all workers have finished.
      while (activeWorkers > 0)
         {
         // Receive an empty message from any worker.
         CommStatus status =
            world.receive
               (null, MASTER_MSG, IntegerBuf.emptyBuffer());
         worker = status.fromRank;

         // Send next chunk range to that specific worker. If null,
         // no more work.
         range = schedule.next (worker);
         world.send (worker, WORKER_MSG, ObjectBuf.buffer (range));
         if (range == null) -- activeWorkers;
         }
      }

   /**
    * Perform the worker section.
    */
   private static void workerSection()
      throws IOException
      {
      // Process chunks from master.
      for (;;)
         {
```

```
// Receive chunk range from master. If null, no more work.
ObjectItemBuf<Range> rangeBuf = ObjectBuf.buffer();
world.receive (0, WORKER_MSG, rangeBuf);
Range range = rangeBuf.item;
if (range == null) break;
int lb = range.lb();
int ub = range.ub();
++ chunkCount;

// Compute all rows and columns in slice.
for (int r = lb; r <= ub; ++ r)
   {
   double y = ycenter + (yoffset - r) / resolution;

   for (int c = 0; c < width; ++ c)
      {
      double x = xcenter + (xoffset + c) / resolution;

      // Iterate until convergence.
      int i = 0;
      double aold = 0.0;
      double bold = 0.0;
      double a = 0.0;
      double b = 0.0;
      double zmagsqr = 0.0;
      while (i < maxiter && zmagsqr <= 4.0)
         {
         ++ i;
         a = aold*aold - bold*bold + x;
         b = 2.0*aold*bold + y;
         zmagsqr = a*a + b*b;
         aold = a;
         bold = b;
         }

      // Increment histogram counter for pixel value.
      ++ histogram[i];
      }
   }
```

We tag the chunk-finished messages with MASTER_MSG so the master will not confuse them with the reduction messages.

```
            // Report completion of slice to master.
            world.send (0, MASTER_MSG, IntegerBuf.emptyBuffer());
            }
         };
      }
```

Table 26.2 (at the end of the chapter) lists, and Figure 26.7 plots, the MSHistogramClu program's performance on the "tardis" parallel computer, using a dynamic schedule with a chunk size of 10. As did the PiClu program, the MSHistogramClu program achieves excellent speedups and efficiencies.

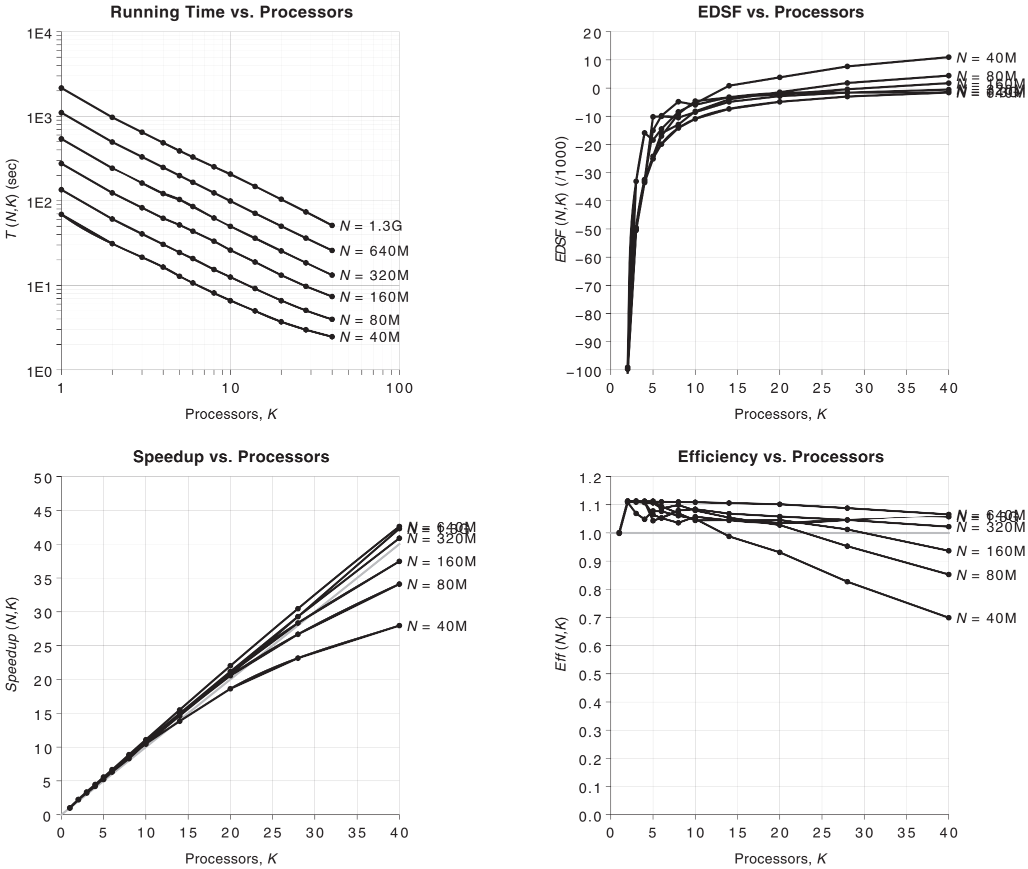


**Figure 26.7** MSHistogramSeq/MSHistogramClu running-time metrics

**Table 26.1** PiSeq/PiClu running-time metrics

| *N* | *K* | *T* | *Spdup* | *Eff* | *EDSF* | *N* | *K* | *T* | *Spdup* | *Eff* | *EDSF* |
|---:|---:|---:|---:|---:|---:|---:|---:|---:|---:|---:|---:|
| 1G | seq | 67170 | | | | 10G | seq | 671429 | | | |
| 1G | 1 | 67193 | 1.000 | 1.000 | | 10G | 1 | 671445 | 1.000 | 1.000 | |
| 1G | 2 | 33612 | 1.998 | 0.999 | 0.000 | 10G | 2 | 335764 | 2.000 | 1.000 | 0.000 |
| 1G | 3 | 22432 | 2.994 | 0.998 | 0.001 | 10G | 3 | 223864 | 2.999 | 1.000 | 0.000 |
| 1G | 4 | 16838 | 3.989 | 0.997 | 0.001 | 10G | 4 | 167979 | 3.997 | 0.999 | 0.000 |
| 1G | 5 | 13483 | 4.982 | 0.996 | 0.001 | 10G | 5 | 134389 | 4.996 | 0.999 | 0.000 |
| 1G | 6 | 11243 | 5.974 | 0.996 | 0.001 | 10G | 6 | 112008 | 5.994 | 0.999 | 0.000 |
| 1G | 8 | 8188 | 8.203 | 1.025 | -0.004 | 10G | 8 | 81343 | 8.254 | 1.032 | -0.004 |
| 1G | 10 | 6590 | 10.193 | 1.019 | -0.002 | 10G | 10 | 65027 | 10.325 | 1.033 | -0.004 |
| 1G | 14 | 4779 | 14.055 | 1.004 | 0.000 | 10G | 14 | 46559 | 14.421 | 1.030 | -0.002 |
| 1G | 20 | 3378 | 19.885 | 0.994 | 0.000 | 10G | 20 | 32647 | 20.566 | 1.028 | -0.001 |
| 1G | 28 | 2481 | 27.074 | 0.967 | 0.001 | 10G | 28 | 23555 | 28.505 | 1.018 | -0.001 |
| 1G | 40 | 1934 | 34.731 | 0.868 | 0.004 | 10G | 40 | 16654 | 40.316 | 1.008 | 0.000 |
| 2G | seq | 134305 | | | | 20G | seq | 1342783 | | | |
| 2G | 1 | 130043 | 1.033 | 1.033 | | 20G | 1 | 1342811 | 1.000 | 1.000 | |
| 2G | 2 | 67186 | 1.999 | 1.000 | 0.033 | 20G | 2 | 671459 | 2.000 | 1.000 | 0.000 |
| 2G | 3 | 44831 | 2.996 | 0.999 | 0.017 | 20G | 3 | 447905 | 2.998 | 0.999 | 0.000 |
| 2G | 4 | 33627 | 3.994 | 0.998 | 0.011 | 20G | 4 | 335765 | 3.999 | 1.000 | 0.000 |
| 2G | 5 | 26922 | 4.989 | 0.998 | 0.009 | 20G | 5 | 268751 | 4.996 | 0.999 | 0.000 |
| 2G | 6 | 22438 | 5.986 | 0.998 | 0.007 | 20G | 6 | 223844 | 5.999 | 1.000 | 0.000 |
| 2G | 8 | 16323 | 8.228 | 1.028 | 0.001 | 20G | 8 | 162631 | 8.257 | 1.032 | -0.004 |
| 2G | 10 | 13085 | 10.264 | 1.026 | 0.001 | 20G | 10 | 129967 | 10.332 | 1.033 | -0.004 |
| 2G | 14 | 9421 | 14.256 | 1.018 | 0.001 | 20G | 14 | 93048 | 14.431 | 1.031 | -0.002 |
| 2G | 20 | 6599 | 20.352 | 1.018 | 0.001 | 20G | 20 | 65273 | 20.572 | 1.029 | -0.001 |
| 2G | 28 | 4892 | 27.454 | 0.981 | 0.002 | 20G | 28 | 46850 | 28.661 | 1.024 | -0.001 |
| 2G | 40 | 3575 | 37.568 | 0.939 | 0.003 | 20G | 40 | 33092 | 40.577 | 1.014 | 0.000 |
| 5G | seq | 335714 | | | | 50G | seq | 3356877 | | | |
| 5G | 1 | 327927 | 1.024 | 1.024 | | 50G | 1 | 3358537 | 1.000 | 1.000 | |
| 5G | 2 | 167948 | 1.999 | 0.999 | 0.024 | 50G | 2 | 1678521 | 2.000 | 1.000 | 0.000 |
| 5G | 3 | 112003 | 2.997 | 0.999 | 0.012 | 50G | 3 | 1119616 | 2.998 | 0.999 | 0.000 |
| 5G | 4 | 84004 | 3.996 | 0.999 | 0.008 | 50G | 4 | 839794 | 3.997 | 0.999 | 0.000 |
| 5G | 5 | 67219 | 4.994 | 0.999 | 0.006 | 50G | 5 | 671848 | 4.996 | 0.999 | 0.000 |
| 5G | 6 | 56026 | 5.992 | 0.999 | 0.005 | 50G | 6 | 559840 | 5.996 | 0.999 | 0.000 |
| 5G | 8 | 40696 | 8.249 | 1.031 | -0.001 | 50G | 8 | 406491 | 8.258 | 1.032 | -0.005 |
| 5G | 10 | 32525 | 10.322 | 1.032 | -0.001 | 50G | 10 | 325256 | 10.321 | 1.032 | -0.004 |
| 5G | 14 | 23356 | 14.374 | 1.027 | 0.000 | 50G | 14 | 232447 | 14.441 | 1.032 | -0.002 |
| 5G | 20 | 16361 | 20.519 | 1.026 | 0.000 | 50G | 20 | 162812 | 20.618 | 1.031 | -0.002 |
| 5G | 28 | 11841 | 28.352 | 1.013 | 0.000 | 50G | 28 | 116786 | 28.744 | 1.027 | -0.001 |
| 5G | 40 | 8417 | 39.885 | 0.997 | 0.001 | 50G | 40 | 81987 | 40.944 | 1.024 | -0.001 |

**Table 26.2** MSHistogramSeq/MSHistogramClu running-time metrics

| *N* | *K* | *T* | *Spdup* | *Eff* | *EDSF* | *N* | *K* | *T* | *Spdup* | *Eff* | *EDSF* |
|---:|---:|---:|---:|---:|---:|---:|---:|---:|---:|---:|---:|
| 40M | seq | 69004 | | | | 320M | seq | 540741 | | | |
| 40M | 1 | 69121 | 0.998 | 0.998 | | 320M | 1 | 540456 | 1.001 | 1.001 | |
| 40M | 2 | 31137 | 2.216 | 1.108 | -0.099 | 320M | 2 | 243135 | 2.224 | 1.112 | -0.100 |
| 40M | 3 | 21516 | 3.207 | 1.069 | -0.033 | 320M | 3 | 162213 | 3.334 | 1.111 | -0.050 |
| 40M | 4 | 16456 | 4.193 | 1.048 | -0.016 | 320M | 4 | 121759 | 4.441 | 1.110 | -0.033 |
| 40M | 5 | 12804 | 5.389 | 1.078 | -0.018 | 320M | 5 | 103687 | 5.215 | 1.043 | -0.010 |
| 40M | 6 | 10685 | 6.458 | 1.076 | -0.014 | 320M | 6 | 85554 | 6.320 | 1.053 | -0.010 |
| 40M | 8 | 8131 | 8.487 | 1.061 | -0.008 | 320M | 8 | 62598 | 8.638 | 1.080 | -0.010 |
| 40M | 10 | 6574 | 10.497 | 1.050 | -0.005 | 320M | 10 | 49885 | 10.840 | 1.084 | -0.009 |
| 40M | 14 | 4991 | 13.826 | 0.988 | 0.001 | 320M | 14 | 36152 | 14.957 | 1.068 | -0.005 |
| 40M | 20 | 3706 | 18.620 | 0.931 | 0.004 | 320M | 20 | 25551 | 21.163 | 1.058 | -0.003 |
| 40M | 28 | 2981 | 23.148 | 0.827 | 0.008 | 320M | 28 | 18462 | 29.289 | 1.046 | -0.002 |
| 40M | 40 | 2468 | 27.959 | 0.699 | 0.011 | 320M | 40 | 13228 | 40.879 | 1.022 | -0.001 |
| 80M | seq | 135252 | | | | 640M | seq | 1102720 | | | |
| 80M | 1 | 135335 | 0.999 | 0.999 | | 640M | 1 | 1102994 | 1.000 | 1.000 | |
| 80M | 2 | 60920 | 2.220 | 1.110 | -0.100 | 640M | 2 | 495872 | 2.224 | 1.112 | -0.101 |
| 80M | 3 | 40644 | 3.328 | 1.109 | -0.050 | 640M | 3 | 330761 | 3.334 | 1.111 | -0.050 |
| 80M | 4 | 30540 | 4.429 | 1.107 | -0.032 | 640M | 4 | 248224 | 4.442 | 1.111 | -0.033 |
| 80M | 5 | 24447 | 5.532 | 1.106 | -0.024 | 640M | 5 | 198555 | 5.554 | 1.111 | -0.025 |
| 80M | 6 | 20744 | 6.520 | 1.087 | -0.016 | 640M | 6 | 165537 | 6.661 | 1.110 | -0.020 |
| 80M | 8 | 15386 | 8.791 | 1.099 | -0.013 | 640M | 8 | 124202 | 8.878 | 1.110 | -0.014 |
| 80M | 10 | 12529 | 10.795 | 1.080 | -0.008 | 640M | 10 | 99457 | 11.087 | 1.109 | -0.011 |
| 80M | 14 | 9165 | 14.757 | 1.054 | -0.004 | 640M | 14 | 71220 | 15.483 | 1.106 | -0.007 |
| 80M | 20 | 6580 | 20.555 | 1.028 | -0.001 | 640M | 20 | 50053 | 22.031 | 1.102 | -0.005 |
| 80M | 28 | 5070 | 26.677 | 0.953 | 0.002 | 640M | 28 | 36204 | 30.459 | 1.088 | -0.003 |
| 80M | 40 | 3967 | 34.094 | 0.852 | 0.004 | 640M | 40 | 25873 | 42.620 | 1.066 | -0.002 |
| 160M | seq | 276031 | | | | 1.3G | seq | 2163755 | | | |
| 160M | 1 | 275835 | 1.001 | 1.001 | | 1.3G | 1 | 2161902 | 1.001 | 1.001 | |
| 160M | 2 | 124171 | 2.223 | 1.111 | -0.100 | 1.3G | 2 | 971695 | 2.227 | 1.113 | -0.101 |
| 160M | 3 | 82831 | 3.332 | 1.111 | -0.050 | 1.3G | 3 | 647868 | 3.340 | 1.113 | -0.050 |
| 160M | 4 | 62227 | 4.436 | 1.109 | -0.033 | 1.3G | 4 | 486099 | 4.451 | 1.113 | -0.034 |
| 160M | 5 | 51853 | 5.323 | 1.065 | -0.015 | 1.3G | 5 | 388923 | 5.563 | 1.113 | -0.025 |
| 160M | 6 | 43716 | 6.314 | 1.052 | -0.010 | 1.3G | 6 | 329667 | 6.563 | 1.094 | -0.017 |
| 160M | 8 | 33314 | 8.286 | 1.036 | -0.005 | 1.3G | 8 | 252850 | 8.557 | 1.070 | -0.009 |
| 160M | 10 | 26105 | 10.574 | 1.057 | -0.006 | 1.3G | 10 | 207235 | 10.441 | 1.044 | -0.005 |
| 160M | 14 | 18882 | 14.619 | 1.044 | -0.003 | 1.3G | 14 | 147804 | 14.639 | 1.046 | -0.003 |
| 160M | 20 | 13198 | 20.915 | 1.046 | -0.002 | 1.3G | 20 | 104506 | 20.705 | 1.035 | -0.002 |
| 160M | 28 | 9739 | 28.343 | 1.012 | 0.000 | 1.3G | 28 | 73890 | 29.283 | 1.046 | -0.002 |
| 160M | 40 | 7369 | 37.458 | 0.936 | 0.002 | 1.3G | 40 | 51168 | 42.287 | 1.057 | -0.001 |

# CHAPTER 27

# All-Gather

in which we simulate the motion of antimatter particles; we encounter a program that needs the all-gather message passing operation; and we observe the all-gather operation's effect on a cluster parallel program's performance

# CHAPTER 27 All-Gather

## 27.1 Antiproton Motion

Once the stuff of theoretical physics and science fiction, **antimatter** is becoming a practical reality. For each particle of normal matter, there is an antiparticle of antimatter. The negatively charged electron's counterpart is the positively charged **positron**. The positively charged proton's antimatter twin is the **antiproton**, which has the same mass as the proton, but the opposite (negative) charge. Particle accelerators, used in particle physics research, produce antimatter routinely. Using the accelerator to generate a high-energy beam of protons and smashing them into a tungsten or copper target liberates a flood of elementary particles, some of which are antiprotons. The antiprotons can then be captured, stored, and used.

You might have heard of **positron emission tomography (PET) scanning**, a medical imaging technique that uses antimatter (positrons rather than antiprotons). Other applications for antimatter include cancer therapy and space propulsion—not by powering the starship Enterprise's warp drive, but by using the antimatter to heat propellant in a conventional rocket.

When an antiproton encounters a normal proton, the two particles instantly annihilate each other, converting their mass completely to energy according to Einstein's formula $E = mc^2$. One therefore cannot simply put antiprotons in a bottle like one would with normal matter; the antiprotons would annihilate the protons in the bottle and would quickly vanish. An effective antiproton trap consists of a high-vacuum vessel (so antiprotons and air molecules will not annihilate each other) plus magnetic fields to confine the antiprotons.

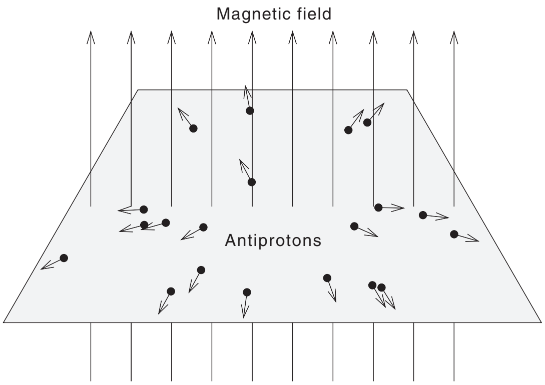


**Figure 27.1** Antiproton trap

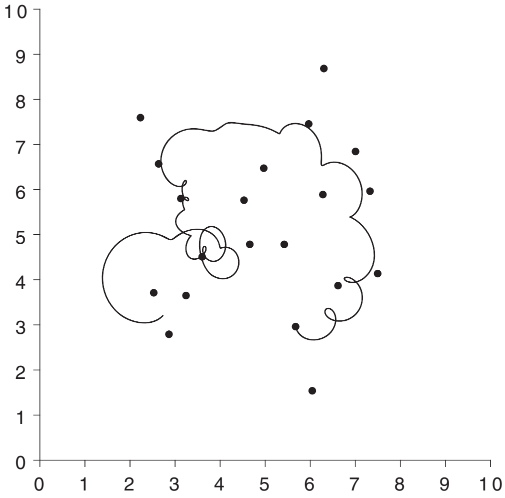


**Figure 27.2** Antiproton track

As a simplified example of an antiproton trap, suppose the antiprotons lie in a two-dimensional plane (Figure 27.1). A uniform magnetic field is applied perpendicular to the plane. The antiprotons, having like charges, repel each other, causing them to move outward in the plane. But as they move, the magnetic field causes their paths to bend back toward the center. Each antiproton follows a complicated jiggling path, alternately being repelled by close encounters with other antiprotons and circling back due to the magnetic field (Figure 27.2). If the magnetic field is strong enough, the antiprotons never get far from the center of the trap and never get the chance to annihilate the protons in the trap walls.

We are going to write a program to simulate the motion of several antiprotons in an antiproton trap, under the influence of their mutual repulsion and the magnetic field. The program places the antiprotons at random starting positions, and then calculates the antiprotons' tracks as a function of time. Figure 27.2 shows the final positions of 20 antiprotons after a certain amount of simulated time has elapsed, as well as the track of one antiproton.

To avoid getting into the quantum mechanical formulas that are needed for a realistic antiproton simulation, we will treat the antiprotons as idealized charged point particles. We begin with a review of the physics of charged particle motion. Our formulas use **two-dimensional (2-D) vectors**. A vector is written with a bold symbol, such as **a**. A 2-D vector consists of an ***x* component** and a ***y* component**. The vector **a** can be written as a pair of components $(a_x, a_y)$.

We need to do arithmetic with vectors. Let $\mathbf{a} = (a_x, a_y)$ and $\mathbf{b} = (b_x, b_y)$ be vectors, and $c$ be a scalar value. Here are the formulas for vector arithmetic as well as the magnitude of a vector:

$$\mathbf{a} + \mathbf{b} = (a_x + b_x,\ a_y + b_y) \qquad \text{Vector sum} \qquad (27.1)$$

$$\mathbf{a} - \mathbf{b} = (a_x - b_x,\ a_y - b_y) \qquad \text{Vector difference} \qquad (27.2)$$

$$c \cdot \mathbf{a} = (c \cdot a_x,\ c \cdot a_y) \qquad \text{Scalar product} \qquad (27.3)$$

$$\left|\mathbf{a}\right| = \sqrt{a_x^2 + a_y^2} \qquad \text{Vector magnitude} \qquad (27.4)$$

A particle's position as a function of time is represented as a 2-D vector $\mathbf{p}(t)$, whose $x$ and $y$ components are $p_x(t)$ and $p_y(t)$:

$$\mathbf{p}(t) = (p_x(t), p_y(t)) \qquad (27.5)$$

The position vector is drawn as an arrow from the origin to the particle's position (Figure 27.3). The components $p_x$ and $p_y$ are the projections of the vector onto the $x$ and $y$ axes.

The time derivative of $\mathbf{p}(t)$ is the particle's velocity vector $\mathbf{v}(t)$:

$$\frac{\mathrm{d}\mathbf{p}(t)}{\mathrm{d}t} = \mathbf{v}(t) = (v_x(t), v_y(t)) \qquad (27.6)$$

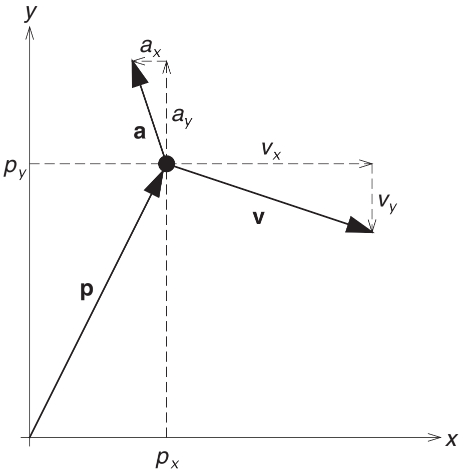


**Figure 27.3** Particle position, velocity, and acceleration vectors

The time derivative of $\mathbf{v}(t)$, which is the second time derivative of $\mathbf{p}(t)$, is the particle's acceleration vector $\mathbf{a}(t)$:

$$\frac{\mathrm{d}^2\mathbf{p}(t)}{\mathrm{d}t^2} = \frac{\mathrm{d}\mathbf{v}(t)}{\mathrm{d}t} = \mathbf{a}(t) = (a_x(t), a_y(t)) \tag{27.7}$$

The acceleration is determined by the force acting on the particle, according to Newton's Second Law of Motion,

$$\mathbf{a}(t) = \frac{\mathbf{f}(t)}{m} \tag{27.8}$$

where $\mathbf{f}(t)$ is the force vector and $m$ is the particle's mass. To simplify the program, we use $m = 1$; then the acceleration is equal to the force. (Or, if you prefer, we use a system of units in which the particle's mass is 1.)

Suppose we know the particle's position at a certain time $t$, and suppose we can determine the particle's acceleration at time $t$. (Soon, we will see how to calculate the force on the particle, that is, the acceleration.) Then a Taylor series expansion lets us calculate the position at time $t$ plus a small increment $\delta$:

$$\mathbf{p}(t+\delta) = \mathbf{p}(t) + \delta \cdot \frac{\mathrm{d}\mathbf{p}(t)}{\mathrm{d}t} + \frac{\delta^2}{2} \cdot \frac{\mathrm{d}^2\mathbf{p}(t)}{\mathrm{d}t^2} + O(\delta^3) \tag{27.9}$$

The $O(\delta^3)$ represents all the remaining terms in the Taylor series. If $\delta$ is small, then these terms (being proportional to $\delta^3$, $\delta^4$, and so on) are much smaller than the first three terms, and we can neglect them. Substituting (27.6) and (27.7) into (27.9) gives the following:

$$\mathbf{p}(t+\delta) = \mathbf{p}(t) + \delta\mathbf{v}(t) + \frac{1}{2}\delta^2\mathbf{a}(t) \tag{27.10}$$

Similarly, a Taylor series expansion for the velocity (omitting the $O(\delta^2)$ and higher terms) is the following:

$$\mathbf{v}(t+\delta) = \mathbf{v}(t) + \delta\mathbf{a}(t) \tag{27.11}$$

These formulas are the basis for an algorithm that calculates the particle's position as a function of time:

```
$\mathbf{p} \leftarrow$ Initial position
$\mathbf{v} \leftarrow$ Initial velocity
for $t = 1$ to *steps*:
    $\mathbf{a} \leftarrow$ Calculate acceleration from $\mathbf{p}$ and $\mathbf{v}$
    $\mathbf{p} \leftarrow \mathbf{p} + \delta\mathbf{v} + 0.5\delta^2\mathbf{a}$
    $\mathbf{v} \leftarrow \mathbf{v} + \delta\mathbf{a}$
```

The algorithm initializes the particle's position and velocity to their values at $t = 0$. The algorithm then does a sequence of **time steps**, calculating the acceleration and updating the position and velocity. Figure 27.4 shows what happens during one time step. At the beginning of time step $t_1$, $\mathbf{p}(t_1)$ and $\mathbf{v}(t_1)$ are the particle's position and velocity. From these, the particle's acceleration at the beginning of the time step, $\mathbf{a}(t_1)$, is calculated. After the update, $\mathbf{p}(t_2)$ and $\mathbf{v}(t_2)$ are the position and velocity at the end of the time step, that is, the beginning of the next time step $t_2 = t_1 + \delta$. From these, the particle's acceleration at the beginning of the next time step, $\mathbf{a}(t_2)$, is calculated. The algorithm thus determines $\mathbf{p}(t)$ at the discrete time values $t = 0$, $\delta$, $2\delta$, $3\delta$, and so on, stopping after a total of *steps* time steps.

The preceding algorithm is an example of a **numerical integration** algorithm. Essentially, it solves the differential equation for $\mathbf{p}(t)$, Equation 27.7, by integrating the right side in a discrete fashion.

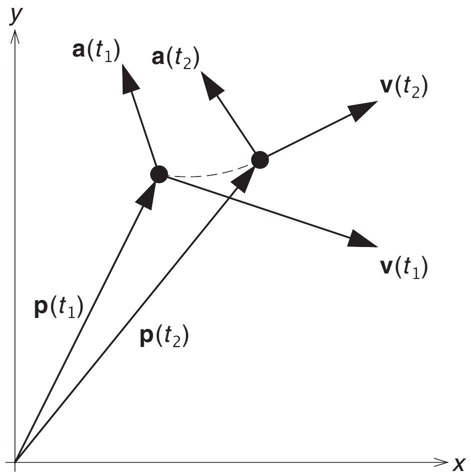


**Figure 27.4** One time step

This algorithm incurs **truncation errors** because of the high-order terms in the Taylor series that are omitted (truncated) from Equations 27.10 and 27.11. To keep the truncation errors small and to determine the positions and velocities accurately, this algorithm must use a very small value for $\delta$. Consequently, this algorithm must take many, many time steps to calculate the particle's motion for an appreciable length of time.

This algorithm is called a **second-order** integration algorithm, because the formula for updating **p** includes terms of order up to and including $\delta$ to the second power. The algorithm is only first order for the velocities. Higher-order integration algorithms do exist. Because their formulas include higher powers of $\delta$, these algorithms can achieve the same truncation error level with a larger value of $\delta$, and can therefore calculate the particle's motion for the same time span with fewer steps. However, in this book, we will stick with the simple second-order integration algorithm.

So far, we have calculated the motion of just one particle. Let's generalize and calculate the motion of *n* particles simultaneously. Now we need arrays of particle positions, velocities, and accelerations, indexed from 0 to *n*–1:

```
for $i = 0$ to $n-1$:
    $\mathbf{p}[i] \leftarrow$ Initial position of particle $i$
    $\mathbf{v}[i] \leftarrow$ Initial velocity of particle $i$
for $t = 1$ to $steps$:
    for $i = 0$ to $n-1$:
        $\mathbf{a}[i] \leftarrow$ Calculate acceleration of particle $i$ from all $\mathbf{p}$s and $\mathbf{v}$s
    for $i = 0$ to $n-1$:
        $\mathbf{p}[i] \leftarrow \mathbf{p}[i] + \delta\mathbf{v}[i] + 0.5\delta^2\mathbf{a}[i]$
        $\mathbf{v}[i] \leftarrow \mathbf{v}[i] + \delta\mathbf{a}[i]$
```

Note that we must calculate all the accelerations first, and only then go back and update all the positions and velocities. This is because the accelerations depend on the positions and velocities, and we don't want the positions and velocities to change until after we've finished calculating the accelerations.

All we need to do now is calculate the acceleration of each particle, which, according to Equation 27.8, is the same as the net force on the particle. The forces on particle *i* include an electrostatic force from every other particle *j* ($j \neq i$) and a magnetic force due to the perpendicular magnetic field. Because force is a vector quantity, we must consider both the direction and the magnitude of each individual force.

Consider first $\mathbf{f}_\mathrm{e}[i,j]$, the electrostatic force on particle *i* from particle *j* (Figure 27.5). Because the particles repel each other, the direction of this force is along the line joining particles *i* and *j*, pointing away from particle *j*. A vector pointing in this direction is the difference between the two position vectors; thus, the force is proportional to the difference vector:

$$\mathbf{f}_\mathrm{e}[i,j] \propto \mathbf{p}[i] - \mathbf{p}[j] \tag{27.12}$$

Let's convert that difference vector to a unit vector (a vector of length 1) pointing in the same direction. Divide the difference vector by its own magnitude. The force is now proportional to the unit vector:

$$\mathbf{f}_\mathrm{e}[i,j] \propto \frac{\mathbf{p}[i] - \mathbf{p}[j]}{\left|\mathbf{p}[i] - \mathbf{p}[j]\right|} \tag{27.13}$$

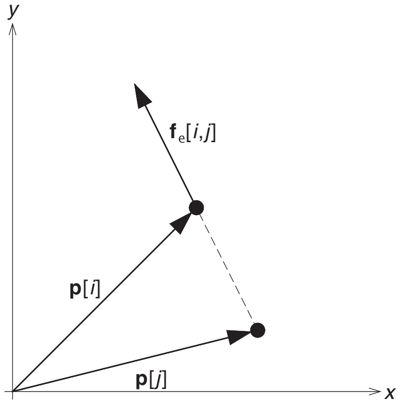


**Figure 27.5** Electrostatic force on particle $i$ due to particle $j$

The actual vector force is that unit vector times the magnitude of the force. The magnitude of the force is equal to the so-called "Coulomb constant," times the product of the two particles' charges, divided by the square of the distance between the two particles—an inverse square law. Because the particles all have the same charge, the product of the Coulomb constant and the particle charges is just another constant that we will call $Q$. The distance between the particles is the magnitude of the difference vector. Thus:

$$\begin{aligned}\mathbf{f}_e[i,j] &= \frac{Q}{\left|\mathbf{p}[i]-\mathbf{p}[j]\right|^2}\cdot\frac{\left|\mathbf{p}[i]-\mathbf{p}[j]\right|}{\left|\mathbf{p}[i]-\mathbf{p}[j]\right|}\\ &= \frac{Q}{\left|\mathbf{p}[i]-\mathbf{p}[j]\right|^3}\cdot(\mathbf{p}[i]-\mathbf{p}[j])\end{aligned} \tag{27.14}$$

The magnetic force on particle $i$ due to the external magnetic field, $\mathbf{f}_m[i]$ (Figure 27.6), is always perpendicular to the particle's velocity. If the only force on the particle is the magnetic force, then the particle moves in a circle. Let the velocity vector be $(v_x[i], v_y[i])$. Then the perpendicular vector is $(v_y[i], -v_x[i])$, and the magnetic force is proportional to this vector. The magnitude of the force is equal to the magnetic field strength, times the particle's charge, times the magnitude of the particle's velocity. The product of the magnetic field strength and the particle charge is a constant that we will call $B$. Thus:

$$\mathbf{f}_m[i] = B(v_y[i], -v_x[i]) \tag{27.15}$$

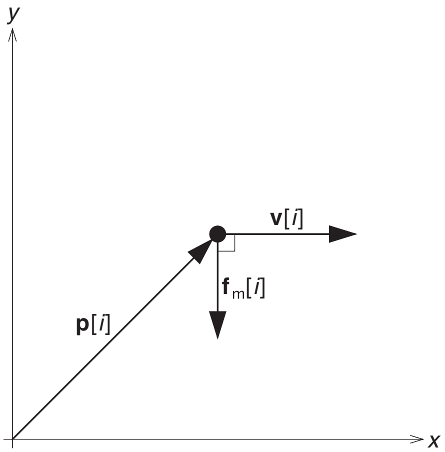


**Figure 27.6** Magnetic force on particle *i*

Whenever a charged particle moves, it creates its own little magnetic field. We will assume that the magnetic fields due to the particle motion are negligible compared to the external magnetic field. We will also assume that the particles never collide; because of their like charges, they repel each other if they come too close.

The net force, and therefore the acceleration, of particle *i* is the vector sum of the electrostatic forces from all other particles *j*, plus the magnetic force. Putting it all together, the algorithm for calculating the particle motions is the following:

```
for $i = 0$ to $n–1$:
    $\mathbf{p}[i] \leftarrow$ Initial position of particle $i$
    $\mathbf{v}[i] \leftarrow$ Initial velocity of particle $i$
for $t = 1$ to $steps$:
    for $i = 0$ to $n–1$:
        $\mathbf{a}[i] \leftarrow \mathbf{0}$
        for $j = 0$ to $n–1, j \neq i$:
            $\mathbf{d} \leftarrow \mathbf{p}[i] - \mathbf{p}[j]$
            $\mathbf{a}[i] \leftarrow \mathbf{a}[i] + \mathbf{d} \cdot Q / |\mathbf{d}|^3$
        $\mathbf{a}[i] \leftarrow \mathbf{a}[i] + B \cdot (v_y[i], -v_x[i])$
    for $i = 0$ to $n–1$:
        $\mathbf{p}[i] \leftarrow \mathbf{p}[i] + \delta\mathbf{v}[i] + 0.5\delta^2\mathbf{a}[i]$
        $\mathbf{v}[i] \leftarrow \mathbf{v}[i] + \delta\mathbf{a}[i]$
```

Now that the basic algorithm is in place, we will rearrange a few things and add some features. First, we initialize the particle positions to random locations in the central portion of the square from 0 to *R* (a parameter) in the $x$ and $y$ directions, and we initialize the particle velocities to zero so the particles are initially at rest:

```
for $i = 0$ to $n - 1$:
    $\mathbf{p}[i] \leftarrow$ (random ($R/4$ .. $3R/4$), random ($R/4$ .. $3R/4$))
    $\mathbf{v}[i] \leftarrow \mathbf{0}$
for $t = 1$ to $steps$:
    for $i = 0$ to $n–1$:
        $\mathbf{a}[i] \leftarrow \mathbf{0}$
        for $j = 0$ to $n–1, j \neq i$:
            $\mathbf{d} \leftarrow \mathbf{p}[i] - \mathbf{p}[j]$
            $\mathbf{a}[i] \leftarrow \mathbf{a}[i] + \mathbf{d} \cdot Q / |\mathbf{d}|^3$
        $\mathbf{a}[i] \leftarrow \mathbf{a}[i] + B \cdot (v_y[i], -v_x[i])$
    for $i = 0$ to $n–1$:
        $\mathbf{p}[i] \leftarrow \mathbf{p}[i] + \delta\mathbf{v}[i] + 0.5\delta^2\mathbf{a}[i]$
        $\mathbf{v}[i] \leftarrow \mathbf{v}[i] + \delta\mathbf{a}[i]$
```

Next, we move the initialization of the acceleration and the computation of the magnetic force from the second loop to the third loop, so that the second loop contains only the electrostatic force calculations. As we will see, this makes it easier to parallelize the program:

```
for $i = 0$ to $n–1$:
    $\mathbf{p}[i] \leftarrow$ (random ($0$ .. $R$), random ($0$ .. $R$))
    $\mathbf{v}[i] \leftarrow \mathbf{0}$
    $\mathbf{a}[i] \leftarrow \mathbf{0}$
for $t = 1$ to $steps$:
    for $i = 0$ to $n–1$:
        for $j = 0$ to $n–1, j \neq i$:
            $\mathbf{d} \leftarrow \mathbf{p}[i] - \mathbf{p}[j]$
            $\mathbf{a}[i] \leftarrow \mathbf{a}[i] + \mathbf{d} \cdot Q / |\mathbf{d}|^3$
    for $i = 0$ to $n–1$:
        $\mathbf{a}[i] \leftarrow \mathbf{a}[i] + B \cdot (v_y[i], -v_x[i])$
        $\mathbf{p}[i] \leftarrow \mathbf{p}[i] + \delta\mathbf{v}[i] + 0.5\delta^2\mathbf{a}[i]$
        $\mathbf{v}[i] \leftarrow \mathbf{v}[i] + \delta\mathbf{a}[i]$
        $\mathbf{a}[i] \leftarrow \mathbf{0}$
```

Let's think about outputting the program's results. For displaying or plotting the particles' motion, we don't have to record the particle positions after each and every time step. Because the second-order integration algorithm requires tiny time steps to achieve acceptable accuracy, the positions change by only minuscule amounts on each time step. Rather, we should take a "snapshot" of the positions only after a certain number of time steps, 1000, say. The algorithm will be controlled by two parameters: we will take *snaps* snapshots, and each snapshot will take place after *steps* time steps. The snapshots will go into an output file:

```
for $i = 0$ to $n-1$:
    $\mathbf{p}[i] \leftarrow$ (random (0 .. $R$), random (0 .. $R$))
    $\mathbf{v}[i] \leftarrow \mathbf{0}$
    $\mathbf{a}[i] \leftarrow \mathbf{0}$
Write snapshot of positions to output file
for $s = 1$ to $snaps$:
    for $t = 1$ to $steps$:
        for $i = 0$ to $n-1$:
            for $j = 0$ to $n-1$, $j \neq i$:
                $\mathbf{d} \leftarrow \mathbf{p}[i] - \mathbf{p}[j]$
                $\mathbf{a}[i] \leftarrow \mathbf{a}[i] + \mathbf{d} \cdot Q \,/\, |\mathbf{d}|^3$
        for $i = 0$ to $n-1$:
            $\mathbf{a}[i] \leftarrow \mathbf{a}[i] + B \cdot (v_y[i], -v_x[i])$
            $\mathbf{p}[i] \leftarrow \mathbf{p}[i] + \delta\mathbf{v}[i] + 0.5\delta^2\mathbf{a}[i]$
            $\mathbf{v}[i] \leftarrow \mathbf{v}[i] + \delta\mathbf{a}[i]$
            $\mathbf{a}[i] \leftarrow \mathbf{0}$
    Write snapshot of positions to output file
```

We need a way to monitor the effects of truncation error in the integration algorithm. One way is to keep track of a **conserved quantity**. One such quantity is the **total momentum**. A particle's momentum is its mass times its velocity; thus, momentum is a vector. Because each particle's mass is 1, in this program, the total momentum is just the vector sum of the velocities. The Law of Conservation of Momentum says that the total momentum of a closed system (one in which there are no external forces) is constant. Because the velocities were all initially zero, the total momentum begins at zero and should stay at zero. At each snapshot, we will compute the total momentum of the system and write that to the output file along with the particle positions. Due to truncation errors in the integration algorithm, as well as roundoff errors from inexact floating point arithmetic, the total momentum will not stay precisely at zero. However, as long as the total momentum is insignificant relative to the particles' velocities, the program's results are acceptably accurate. If the total momentum becomes too large, it's a signal that the program's results are becoming inaccurate. Adding computation of the total momentum **m** to our pseudocode gives the final algorithm that we will implement:

```
for $i = 0$ to $n-1$:
    $\mathbf{p}[i] \leftarrow$ (random (0 .. $R$), random (0 .. $R$))
    $\mathbf{v}[i] \leftarrow \mathbf{0}$
    $\mathbf{a}[i] \leftarrow \mathbf{0}$
$\mathbf{m} \leftarrow \mathbf{0}$
Write snapshot of positions and $\mathbf{m}$ to output file
for $s = 1$ to $snaps$:
    for $t = 1$ to $steps$:
        for $i = 0$ to $n-1$:
```

```
        for $j = 0$ to $n-1$, $j \ne i$:
            $\mathbf{d} \leftarrow \mathbf{p}[i] - \mathbf{p}[j]$
            $[i] \leftarrow \mathbf{a}[i] + \mathbf{d} \cdot Q \,/\, |\mathbf{d}|^3$
    for $i = 0$ to $n-1$:
        $\mathbf{a}[i] \leftarrow \mathbf{a}[i] + B \cdot (v_y[i], -v_x[i])$
        $\mathbf{p}[i] \leftarrow \mathbf{p}[i] + \delta\mathbf{v}[i] + 0.5\delta^2\,\mathbf{a}[i]$
        $\mathbf{v}[i] \leftarrow \mathbf{v}[i] + \delta\mathbf{a}[i]$
        $\mathbf{a}[i] \leftarrow \mathbf{0}$
$\mathbf{m} \leftarrow \mathbf{0}$
for $i = 0$ to $n-1$:
    $\mathbf{m} \leftarrow \mathbf{m} + \mathbf{v}[i]$
Write snapshot of positions and $\mathbf{m}$ to output file
```

## 27.2 Sequential Antiproton Program

Class edu.rit.clu.antimatter.AntiprotonSeq in the Parallel Java Library is a sequential version of the particle motion algorithm. The program takes the following parameters from the command line:

- *seed*—PRNG seed for initializing the particle positions.
- *R*—Side of the square within which the particles are initially positioned.
- *dt*—Size of the time step, $\delta$.
- *steps*—Number of time steps in each snapshot.
- *snaps*—Number of snapshots.
- *n*—Number of particles.
- *outfile*—Output file name.

The data used for Figure 27.2 was generated by the following command:

```
$ java edu.rit.clu.antimatter.AntiprotonSeq \
  142857890 10 0.00001 1000 1000 20 plot_a.dat
```

The output file records the particle position and total momentum snapshots in a binary format. A separate class, class edu.rit.clu.antimatter.AntiprotonFile, handles all the file I/O. Visualization programs can easily be written to display the stored particle motion data in different ways. As an example, Figure 27.2 was generated by this command:

```
$ java edu.rit.clu.antimatter.AntiprotonPlot plot_a.dat
```

The AntiprotonPlot program plots the final positions of all the particles in the file, as well as the track (sequence of positions) of the first particle in the file.

The AntiprotonSeq program makes extensive use of 2-D vectors. Class edu.rit.vector.Vector2D encapsulates a 2-D vector and provides methods for vector arithmetic, vector magnitude, and other operations.

Here is the source code for the AntiprotonSeq program.

```
package edu.rit.clu.antimatter;
import edu.rit.util.Random;
import edu.rit.util.Range;
import edu.rit.vector.Vector2D;
import java.io.BufferedOutputStream;
import java.io.File;
import java.io.FileOutputStream;
public class AntiprotonSeq
   {
   // Charge on an antiproton.
   static final double QP = 3.0;

   // Magnetic field strength.
   static final double B = 3.0;

   static final double QP_QP = QP * QP;
   static final double QP_B = QP * B;

   // Command line arguments.
   static long seed;
   static double R;
   static double dt;
   static int steps;
   static int snaps;
   static int N;
   static File outfile;

   static double one_half_dt_sqr;

   // Acceleration, velocity, and position vector arrays.
   static Vector2D[] a;
   static Vector2D[] v;
   static Vector2D[] p;

   // Total momentum.
   static Vector2D totalMV = new Vector2D();

   // Temporary storage.
   static Vector2D temp = new Vector2D();

   /**
    * Main program.
    */
   public static void main
```

```
(String[] args)
throws Exception
{
// Start timing.
long t1 = System.currentTimeMillis();

// Parse command line arguments.
if (args.length != 7) usage();
seed = Long.parseLong (args[0]);
R = Double.parseDouble (args[1]);
dt = Double.parseDouble (args[2]);
steps = Integer.parseInt (args[3]);
snaps = Integer.parseInt (args[4]);
N = Integer.parseInt (args[5]);
outfile = new File (args[6]);
```

To reduce the running time, we'll compute $0.5\delta^2$ once here, instead of repeatedly in the loop later on.

```
one_half_dt_sqr = 0.5 * dt * dt;

// Create pseudorandom number generator.
Random prng = Random.getInstance (seed);

// Initialize acceleration, velocity, and position vector
// arrays.
a = new Vector2D [N];
v = new Vector2D [N];
p = new Vector2D [N];
for (int i = 0; i < N; ++ i)
   {
   a[i] = new Vector2D();
   v[i] = new Vector2D();
   p[i] = new Vector2D
      (prng.nextDouble()*R/2+R/4, prng.nextDouble()*R/2+R/4);
   }

// Set up output file and write initial snapshot.
AntiprotonFile out =
   new AntiprotonFile (seed, R, dt, steps, snaps+1, N, 0, N);
AntiprotonFile.Writer writer =
   out.prepareToWrite
      (new BufferedOutputStream
         (new FileOutputStream (outfile)));
writer.writeSnapshot (p, 0, totalMV);
```

```
      long t2 = System.currentTimeMillis();

      // Do <snaps> snapshots.
      for (int s = 0; s < snaps; ++ s)
         {
         // Advance time by <steps> steps.
         for (int t = 0; t < steps; ++ t)
            {
            computeAcceleration();
            step();
            }

         // Compute total momentum.
         computeTotalMomentum();

         // Write snapshot.
         writer.writeSnapshot (p, 0, totalMV);
         }

      // Close output file.
      writer.close();

      // Stop timing.
      long t3 = System.currentTimeMillis();
      System.out.println ((t2-t1) + " msec pre");
      System.out.println ((t3-t2) + " msec calc");
      System.out.println ((t3-t1) + " msec total");
      }

   /**
    * Compute the antiproton accelerations due to the repulsive
    * forces from all the antiprotons.
    */
   private static void computeAcceleration()
      {
      // Accumulate forces between each pair of antiprotons, but
      // not between an antiproton and itself.
      for (int i = 0; i < N; ++ i)
         {
         Vector2D a_i = a[i];
         Vector2D p_i = p[i];
```

We must not try to compute the repulsive force between a particle and itself; that is, we must omit the inner loop iteration where $j = i$. It will save time to split the inner loop over $j$ into two parts, one from 0 to $i$–1 and one from $i$+1 to $n$–1. If we wrote a single loop with a test to ensure $j \neq i$, all those tests would increase the running time. Also, note how we compute $d^3$ in the denominator of the force expression, where $d$ is the distance between particles $i$ and $j$. First, we compute $d^2$, and then we compute $d^2 \cdot \sqrt{d^2}$ ; this requires one square root and one multiplication. We could have computed $d = \sqrt{d^2}$ , and then $d{\cdot}d{\cdot}d$, but that requires one square root and two multiplications, which takes longer.

```
         for (int j = 0; j < i; ++ j)
            {
            temp.assign (p_i);
            temp.sub (p[j]);
            double dsqr = temp.sqrMag();
            temp.mul (QP_QP / (dsqr * Math.sqrt(dsqr)));
            a_i.add (temp);
            }
         for (int j = i+1; j < N; ++ j)
            {
            temp.assign (p_i);
            temp.sub (p[j]);
            double dsqr = temp.sqrMag();
            temp.mul (QP_QP / (dsqr * Math.sqrt(dsqr)));
            a_i.add (temp);
            }
         }
      }

   /**
    * Take one time step.
    */
   private static void step()
      {
      // Move all antiprotons.
      for (int i = 0; i < N; ++ i)
         {
         Vector2D a_i = a[i];
         Vector2D v_i = v[i];
         Vector2D p_i = p[i];

         // Accumulate acceleration on antiproton from magnetic
         // field.
         temp.assign (v_i) .mul (QP_B) .rotate270();
         a_i.add (temp);

         // Update antiproton's position and velocity.
         temp.assign (v_i);
         p_i.add (temp.mul (dt));
         temp.assign (a_i);
         p_i.add (temp.mul (one_half_dt_sqr));
         temp.assign (a_i);
         v_i.add (temp.mul (dt));

         // Clear antiproton’s acceleration for the next step.
         a_i.clear();
         }
      }

   /**
    * Compute the total momentum for all the antiprotons. The
    * answer is stored in totalMV.
    */
   private static void computeTotalMomentum()
      {
      totalMV.clear();
      for (int i = 0; i < N; ++ i)
         {
         totalMV.add (v[i]);
         }
      }
   }
```

## 27.3 Collective Communication: All-Gather

Let’s now design a cluster parallel version of the particle motion program. Most of the program’s running time is spent calculating the electrostatic forces between each pair of particles; this portion’s running time is $O(n^2)$ for $n$ particles. The magnetic force calculations, and the position and velocity updates, are only $O(n)$. As the problem size scales up, the latter calculations occupy a diminishing fraction of the total running time. So we will concentrate our attention on the electrostatic force calculations.

We can partition the particles among the $K$ parallel processes, so that each process calculates the positions for $n/K$ particles. Because the calculations take the same amount of time for each particle, this fixed partitioning should result in a balanced load. Let a process’s particle index subrange be *lb* through *ub*. The pseudocode for calculating the forces and updating the positions and velocities in that subrange is:

```
for $i = lb$ to $ub$:
   for $j = 0$ to $n–1$, $j \neq i$:
      $\mathbf{d} \leftarrow \mathbf{p}[i] - \mathbf{p}[j]$
      $\mathbf{a}[i] \leftarrow \mathbf{a}[i] + \mathbf{d} \cdot Q \,/\, |\mathbf{d}|^3$
for $i = lb$ to $ub$:
   $\mathbf{a}[i] \leftarrow \mathbf{a}[i] + B \cdot (v_y[i], -v_x[i])$
   $\mathbf{p}[i] \leftarrow \mathbf{p}[i] + \delta\mathbf{v}[i] + 0.5\delta^2\,\mathbf{a}[i]$
   $\mathbf{v}[i] \leftarrow \mathbf{v}[i] + \delta\mathbf{a}\,[i]$
   $\mathbf{a}[i] \leftarrow \mathbf{0}$
```

While the two outer loops over *i* now range from *lb* to *ub*, the inner loop over *j* still ranges from 0 to *n*–1. Each particle in the process's subrange must still be paired with every other particle to calculate the electrostatic forces.

This has two implications for the cluster parallel program design. First, each process need only set aside storage for its own slice of the velocity array and its own slice of the acceleration array. However, each process must set aside storage for the *entire* position array (Figure 27.7). During one time step, each process reads the entire position array while calculating the electrostatic forces; each process updates only its own slice of the position array with the new particle positions.

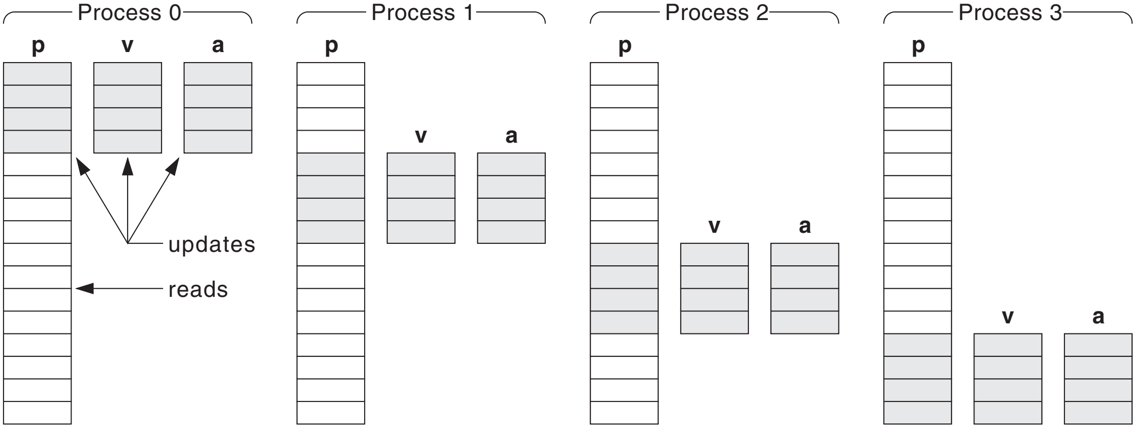


**Figure 27.7** Storage allocation in each process

Second, after updating its own position array slice and before going on to the next time step, each process must fill out the rest of the position array with the new position array slices the other processes have computed. To do so, each process must send its own slice to every other process and must receive a slice from every other process (Figure 27.8). This is precisely the *all-gather* collective communication operation. (Only the positions need to be communicated, not the velocities or accelerations.) The all-gather operation also acts as a synchronization point to enforce the sequential dependency from one time step to the next. The processes will not go on to the next time step until the all-gather has finished and the new particle positions have been disseminated.

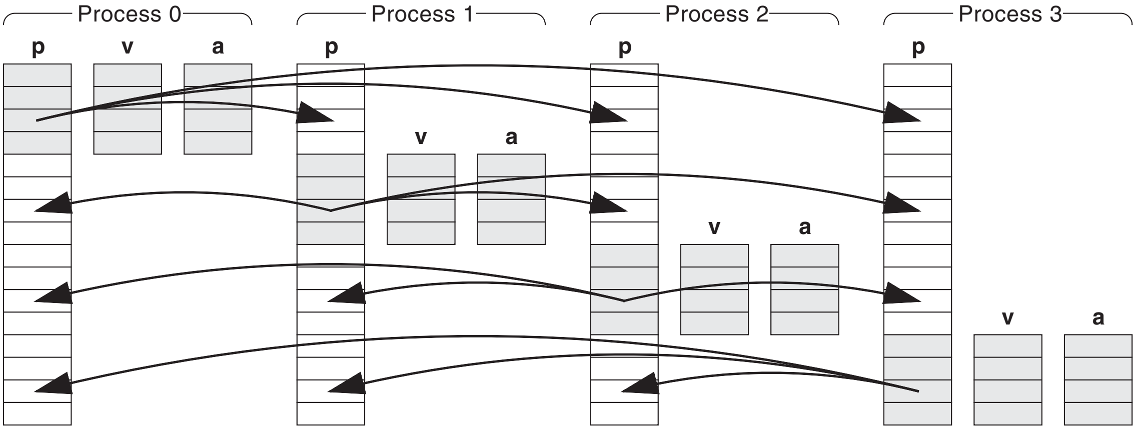


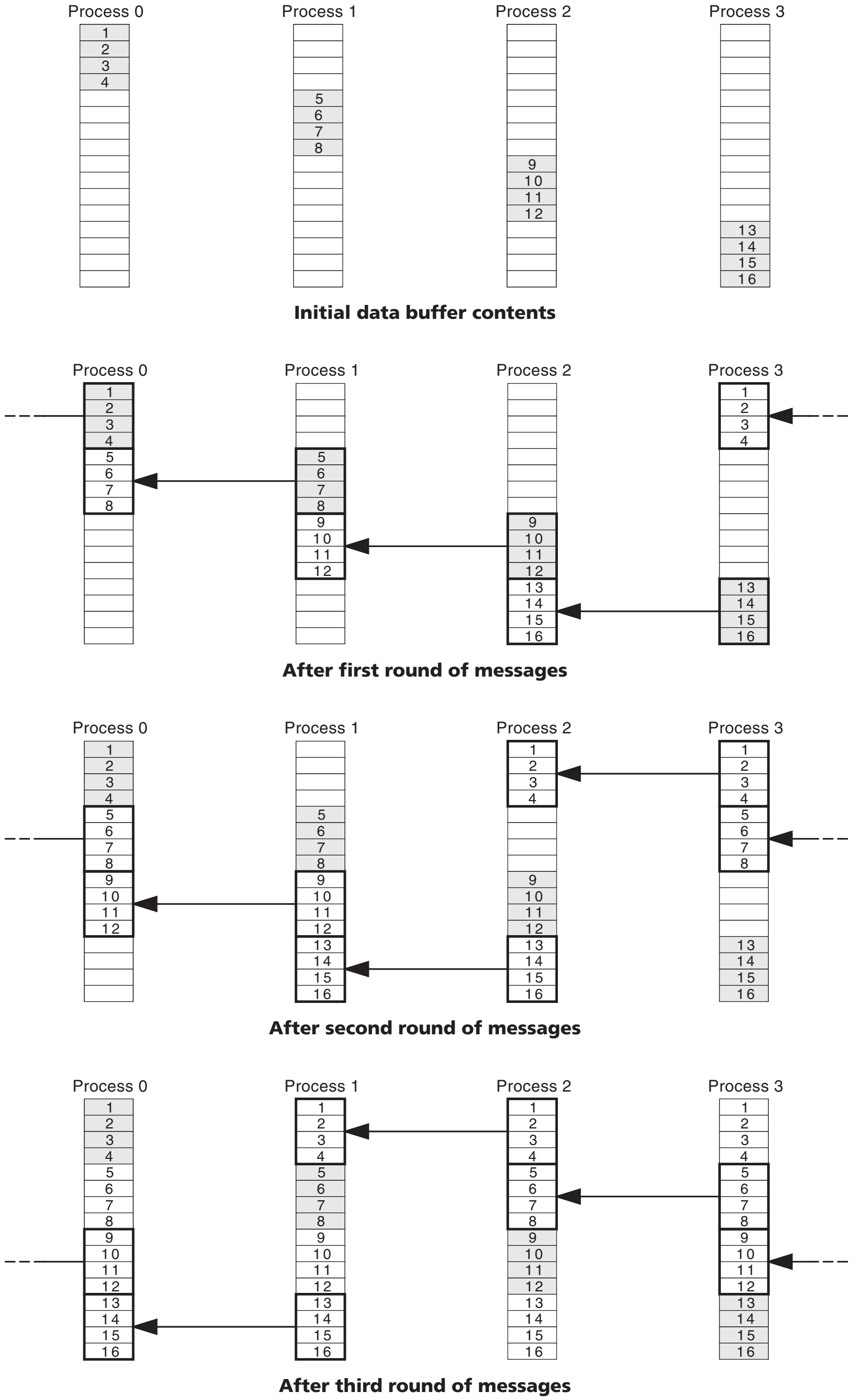
**Figure 27.8** All-gathering the position array slices

Let's consider how long it will take to do an all-gather operation among $K$ processes. Figure 27.9 shows the actual messages the all-gather operation sends, as implemented in the Parallel Java Library (which is targeted at commodity clusters where each backend node has one network interface). During each message round, each process simultaneously sends one data buffer (position array slice) to its predecessor and receives another data buffer from its successor. After $K$–1 message rounds, every process has every slice. Therefore, the time to do an all-gather is the time to send one message multiplied by $K$–1. Using the message send time model (Equation 24.1), the **all-gather time model** for the "tardis" computer is

$$T = (2.08 \times 10^{-4} + 1.07 \times 10^{-9} B)\ (K - 1) \tag{27.16}$$

where $T$ is the all-gather time in seconds, $B$ is the message size in bits (assumed to be the same for every process), and $K$ is the number of processes.

One final design detail is that the cluster parallel version uses the *parallel output files pattern* to improve performance. While one process could write snapshots of all the particle positions into a single output file (because the program has to all-gather the positions into every process anyway), doing so would increase the sequential fraction. Instead, each process writes snapshots of just its own slice into its own separate output file, in parallel with the other processes. Class AntiprotonFile supports writing snapshots of just one slice of the position array.



Initial data buffer contents

After first round of messages

After second round of messages

After third round of messages

**Figure 27.9** Contents of the data buffers during an all-gather operation

## 27.4 Parallel Antiproton Program

Here is the source code for class edu.rit.clu.antimatter.AntiprotonClu, the cluster parallel particle motion program.

```
package edu.rit.clu.antimatter;
import edu.rit.io.Files;
import edu.rit.mp.DoubleBuf;
import edu.rit.pj.Comm;
import edu.rit.util.Random;
import edu.rit.util.Range;
import edu.rit.vector.Vector2D;
import java.io.BufferedOutputStream;
import java.io.File;
import java.io.FileOutputStream;
public class AntiprotonClu
   {
   // Charge on an antiproton.
   static final double QP = 3.0;

   // Magnetic field strength.
   static final double B = 3.0;

   static final double QP_QP = QP * QP;
   static final double QP_B = QP * B;

   // World communicator.
   static Comm world;
   static int size;
   static int rank;

   // Command line arguments.
   static long seed;
   static double R;
   static double dt;
   static int steps;
   static int snaps;
   static int N;
   static File outfile;

   static double one_half_dt_sqr;

   // Antiproton slices.
   static Range[] slices;
   static Range mySlice;
   static int myLb;
   static int myLen;

   // Acceleration, velocity, and position vector arrays.
   static Vector2D[] a;
   static Vector2D[] v;
   static Vector2D[] p;

   // Total momentum.
   static Vector2D totalMV = new Vector2D();

   // Position array communication buffers.
   static DoubleBuf[] buffers;
   static DoubleBuf myBuffer;

   // Temporary storage.
   static Vector2D temp = new Vector2D();

   /**
    * Main program.
    */
   public static void main
      (String[] args)
      throws Exception
      {
      // Start timing.
      long t1 = System.currentTimeMillis();

      // Initialize world communicator.
      Comm.init (args);
      world = Comm.world();
      size = world.size();
      rank = world.rank();

      // Parse command line arguments.
      if (args.length != 7) usage();
      seed = Long.parseLong (args[0]);
      R = Double.parseDouble (args[1]);
      dt = Double.parseDouble (args[2]);
      steps = Integer.parseInt (args[3]);
      snaps = Integer.parseInt (args[4]);
      N = Integer.parseInt (args[5]);
      outfile = new File (args[6]);
      one_half_dt_sqr = 0.5 * dt * dt;
```

We partition the particle index range among the processes.

```
// Set up antiproton slices.
slices = new Range (0, N-1) .subranges (size);
mySlice = slices[rank];
myLb = mySlice.lb();
myLen = mySlice.length();

// Create pseudorandom number generator.
Random prng = Random.getInstance (seed);
```

In each process, we allocate storage for the entire position array, and we generate the same random initial positions (because the PRNG is seeded the same in every process).

```
// Initialize position vector array with all antiprotons.
p = new Vector2D [N];
for (int i = 0; i < N; ++ i)
   {
   p[i] = new Vector2D
      (prng.nextDouble()*R/2+R/4, prng.nextDouble()*R/2+R/4);
   }
```

However, we allocate storage for only one slice of the velocity and acceleration arrays.

```
// Initialize acceleration and velocity vector arrays with a
// slice of antiprotons.
a = new Vector2D [myLen];
v = new Vector2D [myLen];
for (int i = 0; i < myLen; ++ i)
   {
   a[i] = new Vector2D();
   v[i] = new Vector2D();
   }
```

Here are the communication buffers for the all-gather operation (Figure 27.10). `myBuffer` is the source buffer for the data this process sends; it refers to this process's slice of the position array. `buffers` is the array of destination buffers for the data this process receives; each buffer in this array refers to a slice of the position array corresponding to one of the other processes.

```
// Set up position array communication buffers.
buffers = Vector2D.doubleSliceBuffers (p, slices);
myBuffer = buffers[rank];

// Set up output file and write initial snapshot.
```

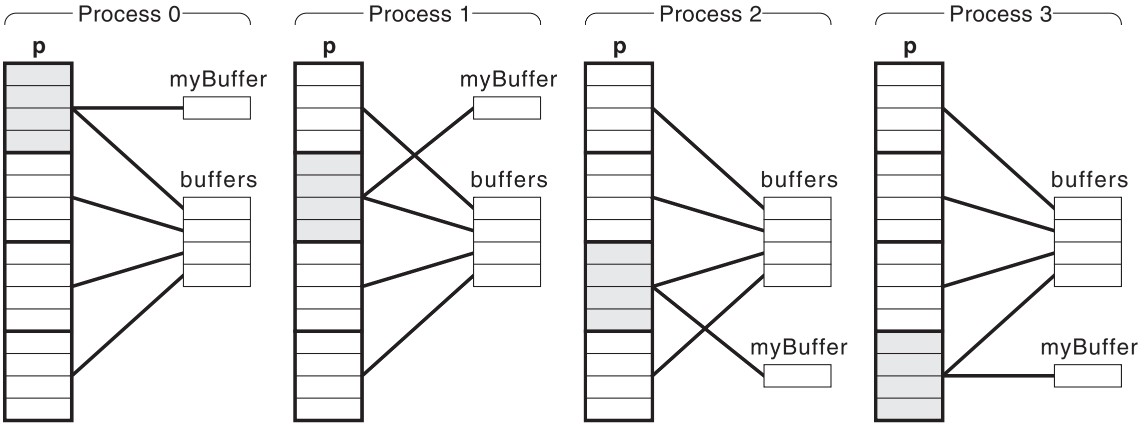


**Figure 27.10** Communication buffers for the all-gather

```
         AntiprotonFile out =
            new AntiprotonFile
               (seed, R, dt, steps, snaps+1, N, myLb, myLen);
         AntiprotonFile.Writer writer =
            out.prepareToWrite
               (new BufferedOutputStream
                  (new FileOutputStream
                     (Files.fileForRank (outfile, rank))));
         writer.writeSnapshot (p, myLb, totalMV);

         long t2 = System.currentTimeMillis();

         // Do <snaps> snapshots.
         for (int s = 0; s < snaps; ++ s)
            {
            // Advance time by <steps> steps.
            for (int t = 0; t < steps; ++ t)
               {
               computeAcceleration();
               step();
```

Here is the all-gather operation at the end of the time step.

```
               // All-gather the new antiproton positions.
               world.allGather (myBuffer, buffers);
               }

            // Compute total momentum.
            computeTotalMomentum();

            // Write snapshot.
            writer.writeSnapshot (p, myLb, totalMV);
            }

         // Close output file.
         writer.close();

         // Stop timing.
         long t3 = System.currentTimeMillis();
         System.out.println ((t2-t1) + " msec pre " + rank);
         System.out.println ((t3-t2) + " msec calc " + rank);
         System.out.println ((t3-t1) + " msec total " + rank);
         }

      /**
       * Compute this process's slice of the antiproton accelerations
       * due to the repulsive forces from all the antiprotons.
       */
      private static void computeAcceleration()
         {
         // Accumulate forces between each pair of antiprotons, but
         // not between an antiproton and itself.
```

The outer loop goes over just the particles in this process's slice.

```
         for (int i = 0; i < myLen; ++ i)
            {
            Vector2D a_i = a[i];
            int index = i + myLb;
            Vector2D p_i = p[index];
```

The inner loop goes over all the particles (except particle *i*).

```
            for (int j = 0; j < index; ++ j)
               {
               temp.assign (p_i);
               temp.sub (p[j]);
               double dsqr = temp.sqrMag();
               temp.mul (QP_QP / (dsqr * Math.sqrt(dsqr)));
               a_i.add (temp);
               }
            for (int j = index+1; j < N; ++ j)
               {
               temp.assign (p_i);
               temp.sub (p[j]);
               double dsqr = temp.sqrMag();
               temp.mul (QP_QP / (dsqr * Math.sqrt(dsqr)));
               a_i.add (temp);
               }
            }
         }

      /**
       * Take one time step.
       */
      private static void step()
         {
         // Move all antiprotons in this slice.
         for (int i = 0; i < myLen; ++ i)
            {
            Vector2D a_i = a[i];
            Vector2D v_i = v[i];
            Vector2D p_i = p[i+myLb];

            // Accumulate acceleration on antiproton from magnetic
            // field.
            temp.assign (v_i) .mul (QP_B) .rotate270();
            a_i.add (temp);

            // Update antiproton’s position and velocity.
            temp.assign (v_i);
            p_i.add (temp.mul (dt));
            temp.assign (a_i);
            p_i.add (temp.mul (one_half_dt_sqr));
            temp.assign (a_i);
            v_i.add (temp.mul (dt));

            // Clear antiproton’s acceleration for the next step.
            a_i.clear();
            }
         }

      /**
       * Compute the total momentum for this process’s slice of the
       * antiprotons. The answer is stored in totalMV.
       */
      private static void computeTotalMomentum()
         {
         totalMV.clear();
         for (int i = 0; i < myLen; ++ i)
            {
            totalMV.add (v[i]);
            }
         }
      }
```

## 27.5 Computation Time Model

Before measuring the cluster parallel particle motion program's performance, let's derive a model for the program's running time. The running time consists of the time to calculate the integration steps plus the time to do all the all-gather operations.

We can determine the time to calculate the integration steps by measuring the sequential version's running time. For $n$ particles and $s$ steps, the sequential version's running time is proportional to $sn^2$. Table 27.1 gives the running-time measurements for the sequential version on the "tardis" computer with $s = 5{,}000$. If we fit the data to the power function model $T_{\text{calc}}(s,n,1) = a{\cdot}sn^2$, we find that $a{\cdot}s = 1.02 \times 10^{-4}$ gives the best fit; thus, $a = 2.04 \times 10^{-8}$. Table 27.1 also shows the $T_{\text{calc}}(s,n,1)$ values computed by the model.

**Table 27.1** Sequential version running times

| $n$ | *Measured* $T_{\text{calc}}(s,n,1)$ (sec) | *Model* $T_{\text{calc}}(s,n,1)$ (sec) |
|---:|---:|---:|
| 1000 | 100.685 | 102.170 |
| 1400 | 200.576 | 200.253 |
| 2000 | 408.967 | 408.680 |
| 2800 | 801.269 | 801.012 |
| 4000 | 1634.195 | 1634.719 |
| 5600 | 3201.878 | 3204.049 |

When we run the parallel program on $K$ processors, the calculation speeds up by a factor of $K$. The parallel program's calculation time therefore is the following:

$$T_{\text{calc}}(s,n,K) = s(2.04 \times 10^{-8} sn^2 \,/\, K) \tag{27.17}$$

For $s$ steps, the parallel version does $s$ all-gathers. Each message in the all-gather consists of one process's slice of the particle position array, that is, $n/K$ position vectors. Each vector consists of two double values, or 128 bits. Plugging this into Equation 27.16, the parallel program's communication time is the following:

$$T_{\text{comm}}(s,n,K) = s(2.08 \times 10^{-4} + 1.37 \times 10^{-7} n \,/\, K)\ (K-1) \tag{27.18}$$

Adding Equations 27.17 and 27.18 together and rearranging terms gives the computation time model for the parallel program,

$$T(s,n,K) = (2.04 \times 10^{-8} sn^2 - 1.37 \times 10^{-7} sn) / K + \\ (2.08 \times 10^{-4} s)K + (1.37 \times 10^{-7} sn - 2.08 \times 10^{-4} s) \quad (27.19)$$

where $s$ is the number of steps, $n$ is the number of particles, $K$ is the number of processors, and $T$ is the running time in seconds. Figure 27.11 plots the calculation time, communication time, and total computation time as a function of $K$ for $s = 5{,}000$ and $n = 2{,}000$.

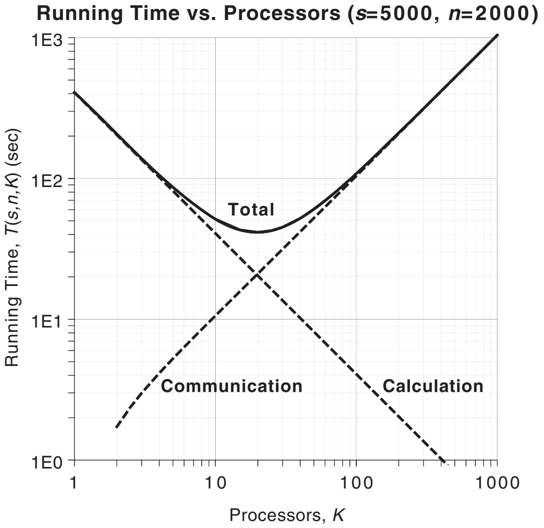


**Figure 27.11** $T(s,n,K)$ predicted by computation time model

Consider what the computation time model tells us about the program's performance as the number of processors $K$ scales up. The model has one term proportional to $1/K$, one term proportional to $K$, and one term that is constant with respect to $K$. For small $K$, the first term dominates, and the running time decreases as $K$ increases. But at some point the second term becomes larger than the first term. After that, the running time starts going up again. This behavior is the same as we observed in the cluster parallel Floyd's Algorithm program in Chapter 25. There, however, the second term was only $O(\log K)$. Here, the second term is $O(K)$, resulting in a much more severe performance reduction as $K$ increases.

The speedup predicted by the computation time model is the following:

$$Speedup(s,n,K) = \frac{T(s,n,1)}{T(s,n,K)} \quad (27.20)$$

As shown in Figure 27.12, the speedup starts out increasing as $K$ increases, but at some point the speedup starts going back down. Like the Floyd's Algorithm program, the computation time model for the cluster parallel particle motion program predicts that the speedup can achieve only a certain maximum value, after which adding more processors causes a *slowdown* rather than a further speedup. This time, however, the slowdown is much more severe. When running this program, we want to use just enough processors to get the maximum speedup.

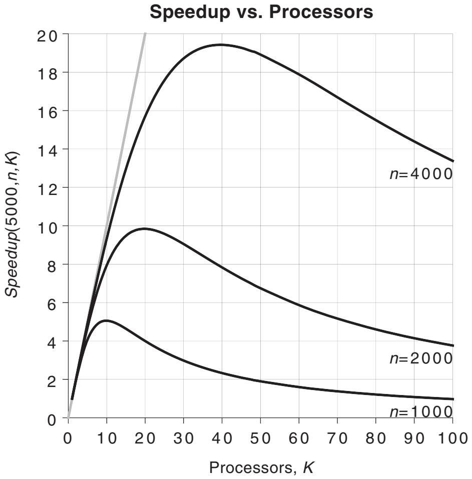


**Figure 27.12** *Speedup*(5000,*n*,*K*) predicted by computation time model

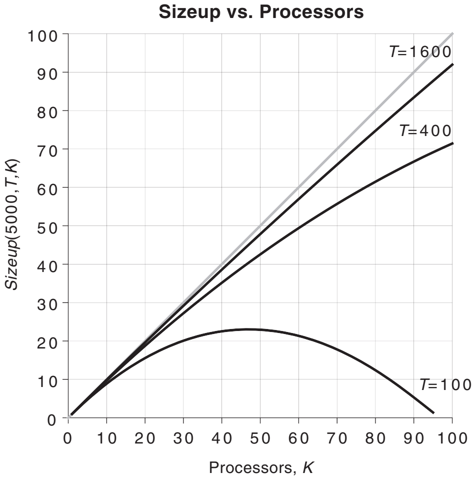


**Figure 27.13** *Sizeup*(5000,*T*,*K*) predicted by computation time model

This point is worth reemphasizing. It's important to run a cluster parallel program with a lot of communication (like this one) on the correct number of processors, so as to achieve the smallest possible running time. Running on the full number of processors in the cluster is not necessarily the right thing to do. Depending on the cluster size and the problem size, this might put you past the "hump" in the speedup curve and cause you to experience a larger running time and a smaller speedup than the optimum. You don't want to run a program for a week on the full cluster, only to discover that you could have had the answer in three days using just part of the cluster. A running-time model is the only way to estimate the performance you'll get, *before* you actually run the program.

To find the maximum speedup, we'll do what we did in Chapter 25: differentiate $T(s,n,K)$ with respect to $K$, set the derivative equal to 0, and solve for $K$. The derivative is the following:

$$\frac{\partial T(s,n,K)}{\partial K} = \frac{1.37\times 10^{-7} sn - 2.04\times 10^{-8} sn^2}{K^2} + 2.08\times 10^{-4} s \tag{27.21}$$

Set the derivative equal to 0 and multiply both sides by $K^2$:

$$1.37\times 10^{-7} sn - 2.04\times 10^{-8} sn^2 + 2.08\times 10^{-4} sK^2 = 0 \tag{27.22}$$

Now solve for $K$:

$$K_{\text{best}} = \sqrt{9.81\times 10^{-5} n^2 - 6.59\times 10^{-4} n} \tag{27.23}$$

For a certain number of particles $n$, Equation 27.23 gives the number of processors $K_{\text{best}}$ that yields the smallest running time, hence the largest speedup. For $n = 2{,}000$ particles, the maximum speedup comes at $K_{\text{best}} \approx 20$. We can get the actual maximum speedup value by plugging $K_{\text{best}}$ into Equation 27.20. For example, $\mathit{Speedup}(5{,}000, 2{,}000, 20) = 9.891$. Notice what Equation 27.23 says about the program's performance as the number of particles $n$ increases. Because $K_{\text{best}}$ is $O(n)$, as the number of particles increases we can scale up to more processors before hitting the slowdown.

What about sizeup? The problem size $N$ is defined so that the amount of computation is proportional to $N$. We'll define $N$ to be the number of electrostatic force calculations, $N = n^2$, neglecting the magnetic force calculations and the position and velocity updates, which are only $O(n)$. Then the sizeup is the following:

$$\mathit{Sizeup}(s,T,K) = \frac{N(s,T,K)}{N(s,T,1)} = \left[\frac{n(s,T,K)}{n(2,T,1)}\right]^2 \tag{27.24}$$

To compute the sizeup, we must first compute the number of particles $n$ that can be solved in time $T$ on $K$ processors with $s$ time steps, $n(s,T,K)$. For a given $s$, $T$, and $K$, Equation 27.19 becomes a quadratic equation in $n$, and solving the quadratic equation yields the value of $n$. (See Appendix C for how to solve a quadratic equation.) Figure 27.13 plots the sizeups predicted for the cluster parallel particle motion program. While less than ideal, the sizeups scale up much better than the speedups as $K$ increases. Consider a problem size of 2,000 particles, which takes about 400 seconds to run on one processor. On 100 processors the sizeup for this running time is about 70, but the speedup for this problem size is only about 4. Once again we see that going for sizeup is preferable to going for speedup. Also notice that for a given running time there is a certain number of processors that gives the maximum sizeup, just as we saw with the speedups.

## 27.6 Parallel Program Performance

Table 27.2 (at the end of the chapter) lists, and Figure 27.14 plots, the AntiprotonClu program's performance on the "tardis" parallel computer. The program was run with the following command,

```
$ java -Dpj.np=$K edu.rit.clu.antimatter.AntiprotonClu 142857 7 \
  0.00001 1000 $SNAPS 1000 outfile.dat
```

where the number of particles was fixed at 1,000, the number of time steps between snapshots was 1,000, and the number of snapshots `($SNAPS)` was 5, 10, 20, 40, 80, or 160. Thus, the total number of time steps varied from 5,000 to 160,000. The shape of the running-time curves is as the running-time model predicts—inversely proportional to $K$ initially, and then becoming directly proportional to $K$. The speedups, efficiencies, and *EDSFs* are the same regardless of the number of snapshots, because the quantity $s$ in the numerator and denominator of the speedup formula (Equation 27.20) cancels out. Thus, speedup, efficiency, and *EDSF* are independent of $s$. The only effect of increasing $s$ is to shift the running-time curves upward, because $T$ is directly proportional to $s$ (Equation 27.19).

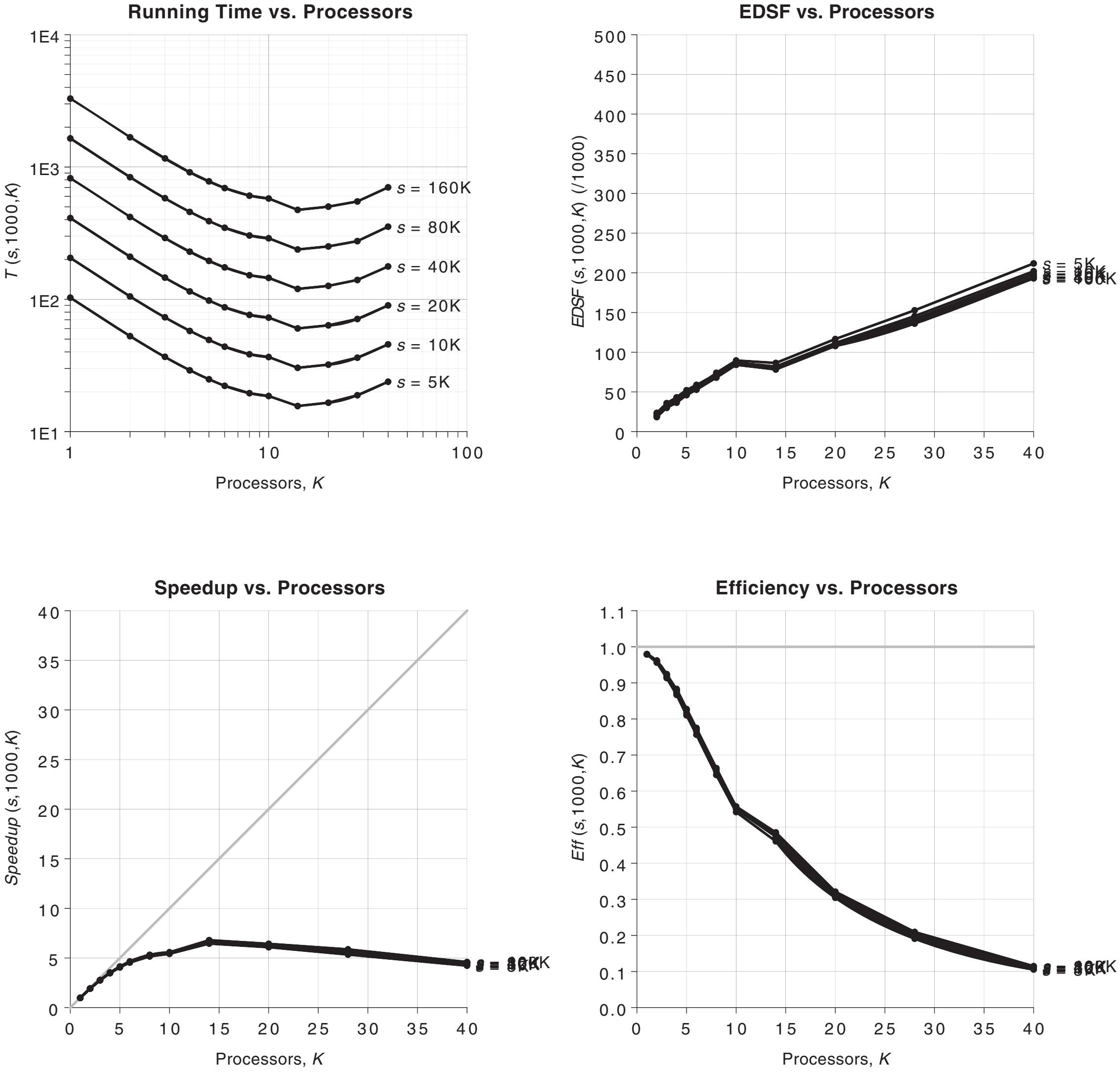


**Figure 27.14** AntiprotonSeq/AntiprotonClu running-time metrics with $s$ varying and $n$ = 1,000

Table 27.3 (at the end of the chapter) lists, and Figure 27.15 plots, the AntiprotonClu program's performance on the "tardis" parallel computer under a different set of conditions. The program was run with the following command,

```
$ java -Dpj.np=$K edu.rit.clu.antimatter.AntiprotonClu 142857 7 \
  0.00001 1000 5 $N outfile.dat
```

where the number of particles `($N)` was 1,000, 1,400, 2,000, 2,800, 4,000, or 5,600, the number of time steps between snapshots was 1,000, and the number of snapshots was 5, for a total of 5,000 time steps.

This time, as the problem size increases, the program's performance gets better—a manifestation of the *surface-to-volume effect.* Considering just the number of particles $n$, the computation time is $O(n^2)$, but the communication time is only $O(n)$. Larger problem sizes lead to diminishing fractions spent on communication.

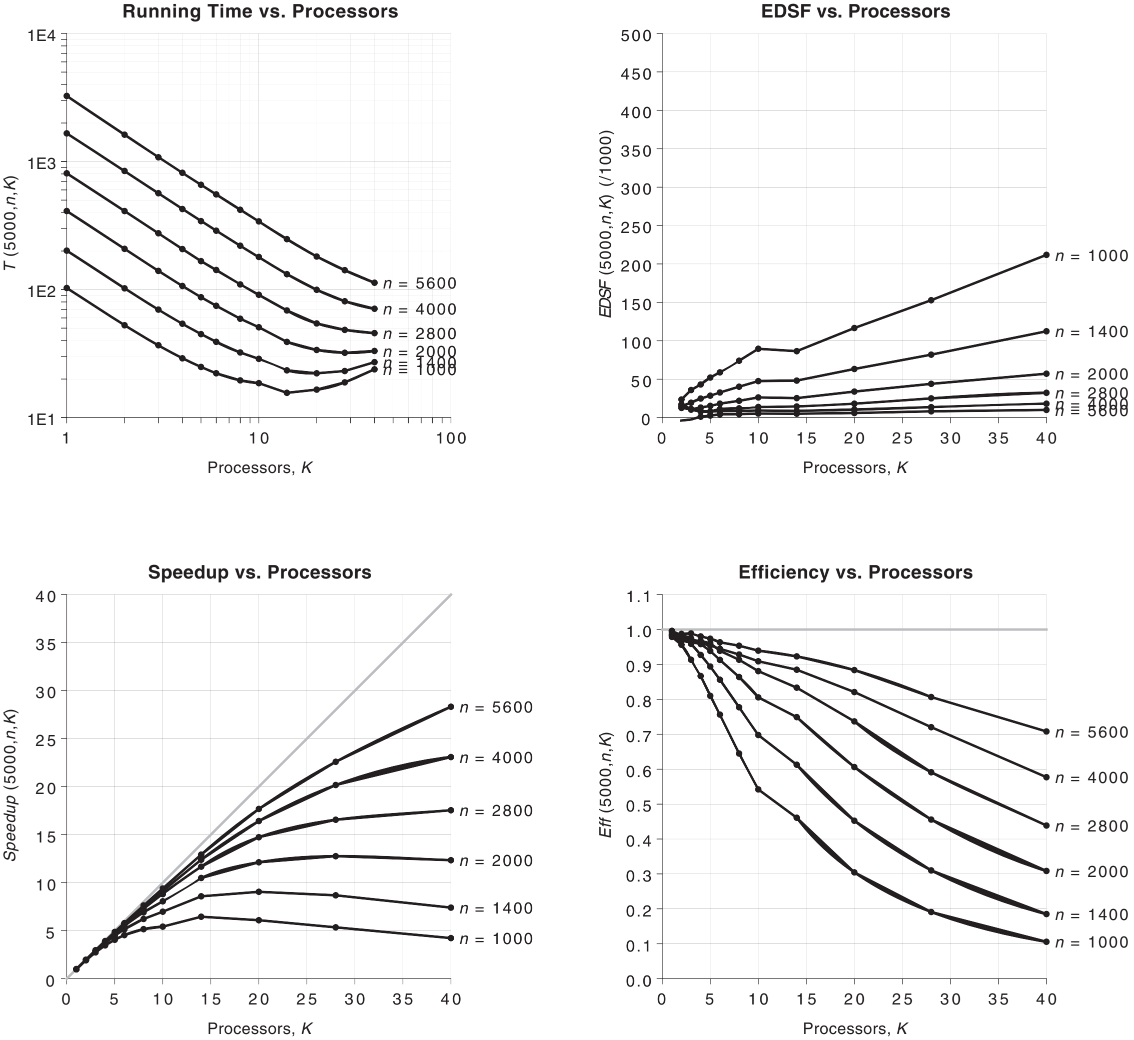


**Figure 27.15** AntiprotonSeq/AntiprotonClu running-time metrics with $s$ = 5,000 and $n$ varying

How well does the model predict the AntiprotonClu program's running time? The communication time model of Equation 27.18 applies when $K$ is 10 processes or fewer, there is one process per node, and all messages are inter-node. When $K$ is 14 or 20, there are two processes on each node, half the messages are between processes on the same node, and half the messages are between processes on different nodes.

In this case, the communication time model is a half-and-half combination of the inter-node message send time model (Equation 24.1) and the intra-node message send time model (Equation 24.2):

$$T_{\text{comm}}(s,n,K) = 1/2\ s\ (2.08\times10^{-4} + 1.37\times10^{-7} n/K)\ (K-1) \\ +\ 1/2\ s\ (7.89\times10^{-5} + 2.89\times10^{-8} n/K)\ (K-1) \tag{27.25}$$

Likewise, when $K$ is 28 or 40, there are four processes on each node, one-quarter of the messages are inter-node, and three-quarters of the messages are intra-node:

$$T_{\text{comm}}(s,n,K) = 1/4\ s\ (2.08\times10^{-4} + 1.37\times10^{-7} n/K)\ (K-1) \\ +\ 3/4\ s\ (7.89\times10^{-5} + 2.89\times10^{-8} n/K)\ (K-1) \tag{27.26}$$

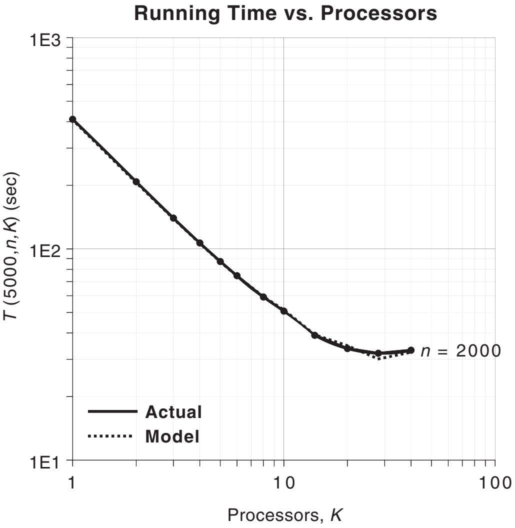


**Figure 27.16** AntiprotonClu predicted and actual running times

Figure 27.16 plots the measured running times, and the predicted running times using the appropriate communication time model, for $n$ = 2,000 particles and $s$ = 5,000 time steps. The predicted time is within 1 percent of the actual time for the majority of the data points and is within 6 percent for all the data points.

To sum up, the particle motion program's running time measurements reinforce what we said in Chapter 24: To get good parallel performance, *the program must have much more computation than communication.* The particle motion program gives better performance than the Floyd's Algorithm program in Chapter 25 because of the particle motion program's more extensive computations. If we increased the problem size by a few more factors beyond what we measured here, we'd expect to see quite reasonable speedups and sizeups.

However, there's still room for improvement. In the next two chapters, we'll change two aspects of the particle motion program's design. First, we'll improve its memory scalability, and then we'll improve (that is, reduce) the time it spends on communication.

## 27.7 The Gravitational *N*-Body Problem

The particle motion program is an example of what physicists call an ***N*-body problem**. While no physicist would actually be interested in calculating the motion of antimatter particles, astrophysicists are interested in a similar problem, the **gravitational *N*-body problem**. In this problem, we are calculating the motion of astronomical bodies, such as planets or stars, due to their mutual gravitational attraction.

When Sir Isaac Newton formulated his laws of gravitation in the seventeenth century, he showed how the motion of two orbiting bodies, like the Sun and the Earth, could be expressed as the formula for a circle, ellipse, parabola, or hyperbola. No such analytic formula exists for a system of three or more bodies. The only way to solve the problem for $N \geq 3$ is to do a numerical integration.

Astronomers have been solving gravitational *N*-body problems for centuries. Before the advent of electronic computing machines, a "computer" was a *human being* who did computation, with pencil and paper, or aided by a mechanical calculating device. There were even flesh-and-blood parallel computers. One of the first gravitational *N*-body computations—and perhaps the first recorded *parallel* computation—took place starting in the spring of 1758. Attempting to predict the exact date of Comet Halley's expected 1759 return, French mathematician Alexis-Claude Clairaut enlisted the help of his two friends, Joseph Lalande and Nicole Lepaute, to compute the comet's orbit as influenced by the gravitational attraction of Jupiter, Saturn, and the sun. Mr. Lalande and Ms. Lepaute calculated the positions of Jupiter and Saturn in their orbits around the sun as a function of time. Mr. Clairaut then took their results and calculated the comet's position as a function of time. In November 1758, after six months of pen-and-paper computations, Mr. Clairaut announced the result: the comet would reach perihelion (the point closest to the sun) on April 15, 1759, plus or minus one month. The comet actually reached perihelion on March 13, 1759. Why the discrepancy? The computers failed to take into account the gravitational influence of Uranus and Neptune—understandably so, because in 1758 those planets had not yet been discovered.

More recently, computational astrophysicists solve gravitational *N*-body problems (on electronic parallel computers) to model the motion of stars in star clusters or galaxies, where *N* may be on the order of $10^5$ or $10^6$. As mentioned in Chapter 1, they use special hardware accelerator chips to do the gravitational force calculations at very high speed.

A gravitational *N*-body program is more complicated than our simple particle motion program. The chief complicating factor is that gravitational forces are attractive rather than repulsive. When two like-charged particles come close, they repel each other and move farther away again. But when two stars come close, they attract each other, move closer still, and start to orbit around each other, forming a so-called "tight binary." The tight binary stars move much more quickly than the other far-apart stars, necessitating small time steps to calculate their motion accurately. However, it would be a waste of processing power to do the force calculations for the far-apart stars on the same tiny time scale as the tight binary stars. Instead, gravitational *N*-body programs use "individual time steps"—a different $\delta$ for each star. Also, to be able to take larger time steps while maintaining acceptable accuracy, gravitational *N*-body programs use much more sophisticated, higher-order integration algorithms than the particle motion program's simple second-order algorithm. A gravitational *N*-body program typically monitors the total energy of the system (gravitational potential energy plus kinetic energy), which is a conserved quantity according to the Law of Conservation of Energy.

Although simplified, the particle motion program does illustrate the features needed in a parallel *N*-body program—the $O(N^2)$ force calculations, the communication of the particle positions at each time step, and computing a conserved quantity to monitor truncation error.

## 27.8 For Further Information

On antimatter and its applications:

- R. Forward. *Indistinguishable from Magic: Speculations and Visions of the Future*. Baen Publishing Enterprises, 1995.
- R. Forward and J. Davis. *Mirror Matter: Pioneering Antimatter Physics*. John Wiley & Sons, 1988.

On the gravitational *N*-body problem:

- P. Hut and J. Makino. The art of computational science. http://www.artcompsci.org/
- S. Aarseth. *Gravitational N-Body Simulations: Tools and Algorithms*. Cambridge University Press, 2003.

On solving very large *N*-body problems with gravitational supercomputers:

- J. Makino, T. Fukushige, M. Koga, and K. Namura. GRAPE-6: massively-parallel special-purpose computer for astrophysical particle simulations. *Publications of the Astronomical Society of Japan*, 55(6):1163–1187, December 2003.
- S. Harfst, A. Gualandris, D. Merritt, R. Spurzem, S. Portegies Zwart, and P. Berczik. Performance analysis of direct *N*-body algorithms on special-purpose supercomputers. *New Astronomy*, 12(5):357–377, July 2007.

On numerical integration methods for solving differential equations:

- W. Press, S. Teukolsky, W. Vetterling, and B. Flannery. *Numerical Recipes: The Art of Scientific Computing, Third Edition*. Cambridge University Press, 2008, Chapter 17.

On Clairaut's, Lalande's, and Lepaute's 1758 calculation of Comet Halley's orbit:

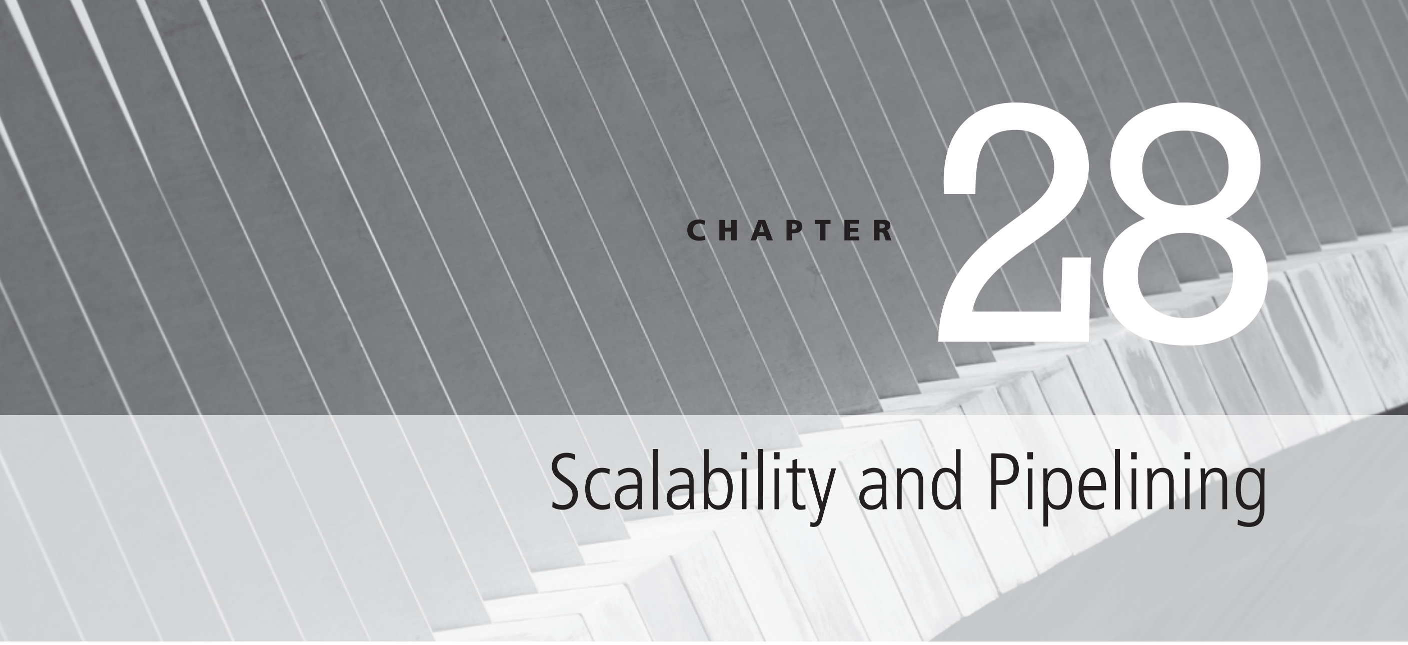
- C. Wilson. Clairaut's calculation of the eighteenth-century return of Halley's Comet. *Journal for the History of Astronomy*, 24:1–14, 1993.
- D. Grier. *When Computers Were Human*. Princeton University Press, 2005.

**Table 27.2** AntiprotonSeq/AntiprotonClu running-time metrics with $s$ varying and $n$ = 1,000

| *s* | *K* | *T* | *Spdup* | *Eff* | *EDSF* | *s* | *K* | *T* | *Spdup* | *Eff* | *EDSF* |
|---:|---:|---:|---:|---:|---:|---:|---:|---:|---:|---:|---:|
| 5000 | seq | 100685 | | | | 40000 | seq | 805411 | | | |
| 5000 | 1 | 102854 | 0.979 | 0.979 | | 40000 | 1 | 822318 | 0.979 | 0.979 | |
| 5000 | 2 | 52639 | 1.913 | 0.956 | 0.024 | 40000 | 2 | 418841 | 1.923 | 0.961 | 0.019 |
| 5000 | 3 | 36744 | 2.740 | 0.913 | 0.036 | 40000 | 3 | 290964 | 2.768 | 0.923 | 0.031 |
| 5000 | 4 | 29036 | 3.468 | 0.867 | 0.043 | 40000 | 4 | 229015 | 3.517 | 0.879 | 0.038 |
| 5000 | 5 | 24858 | 4.050 | 0.810 | 0.052 | 40000 | 5 | 195166 | 4.127 | 0.825 | 0.047 |
| 5000 | 6 | 22186 | 4.538 | 0.756 | 0.059 | 40000 | 6 | 174232 | 4.623 | 0.770 | 0.054 |
| 5000 | 8 | 19517 | 5.159 | 0.645 | 0.074 | 40000 | 8 | 151980 | 5.299 | 0.662 | 0.068 |
| 5000 | 10 | 18575 | 5.420 | 0.542 | 0.090 | 40000 | 10 | 145182 | 5.548 | 0.555 | 0.085 |
| 5000 | 14 | 15604 | 6.453 | 0.461 | 0.086 | 40000 | 14 | 119814 | 6.722 | 0.480 | 0.080 |
| 5000 | 20 | 16535 | 6.089 | 0.304 | 0.117 | 40000 | 20 | 126315 | 6.376 | 0.319 | 0.109 |
| 5000 | 28 | 18828 | 5.348 | 0.191 | 0.153 | 40000 | 28 | 140010 | 5.753 | 0.205 | 0.140 |
| 5000 | 40 | 23815 | 4.228 | 0.106 | 0.212 | 40000 | 40 | 177037 | 4.549 | 0.114 | 0.195 |
| 10000 | seq | 201274 | | | | 80000 | seq | 1611066 | | | |
| 10000 | 1 | 205608 | 0.979 | 0.979 | | 80000 | 1 | 1644748 | 0.980 | 0.980 | |
| 10000 | 2 | 105050 | 1.916 | 0.958 | 0.022 | 80000 | 2 | 837394 | 1.924 | 0.962 | 0.018 |
| 10000 | 3 | 73118 | 2.753 | 0.918 | 0.033 | 80000 | 3 | 581141 | 2.772 | 0.924 | 0.030 |
| 10000 | 4 | 57641 | 3.492 | 0.873 | 0.040 | 80000 | 4 | 457623 | 3.521 | 0.880 | 0.038 |
| 10000 | 5 | 49296 | 4.083 | 0.817 | 0.050 | 80000 | 5 | 389744 | 4.134 | 0.827 | 0.046 |
| 10000 | 6 | 43842 | 4.591 | 0.765 | 0.056 | 80000 | 6 | 347309 | 4.639 | 0.773 | 0.053 |
| 10000 | 8 | 38407 | 5.241 | 0.655 | 0.071 | 80000 | 8 | 303623 | 5.306 | 0.663 | 0.068 |
| 10000 | 10 | 36612 | 5.497 | 0.550 | 0.087 | 80000 | 10 | 289196 | 5.571 | 0.557 | 0.084 |
| 10000 | 14 | 30349 | 6.632 | 0.474 | 0.082 | 80000 | 14 | 237999 | 6.769 | 0.484 | 0.079 |
| 10000 | 20 | 32071 | 6.276 | 0.314 | 0.112 | 80000 | 20 | 251060 | 6.417 | 0.321 | 0.108 |
| 10000 | 28 | 36187 | 5.562 | 0.199 | 0.145 | 80000 | 28 | 275171 | 5.855 | 0.209 | 0.136 |
| 10000 | 40 | 45641 | 4.410 | 0.110 | 0.202 | 80000 | 40 | 353901 | 4.552 | 0.114 | 0.195 |
| 20000 | seq | 402599 | | | | 160000 | seq | 3222247 | | | |
| 20000 | 1 | 411155 | 0.979 | 0.979 | | 160000 | 1 | 3289273 | 0.980 | 0.980 | |
| 20000 | 2 | 209709 | 1.920 | 0.960 | 0.020 | 160000 | 2 | 1677124 | 1.921 | 0.961 | 0.020 |
| 20000 | 3 | 145852 | 2.760 | 0.920 | 0.032 | 160000 | 3 | 1162156 | 2.773 | 0.924 | 0.030 |
| 20000 | 4 | 114716 | 3.510 | 0.877 | 0.039 | 160000 | 4 | 912233 | 3.532 | 0.883 | 0.036 |
| 20000 | 5 | 97814 | 4.116 | 0.823 | 0.047 | 160000 | 5 | 778920 | 4.137 | 0.827 | 0.046 |
| 20000 | 6 | 86984 | 4.628 | 0.771 | 0.054 | 160000 | 6 | 692904 | 4.650 | 0.775 | 0.053 |
| 20000 | 8 | 76198 | 5.284 | 0.660 | 0.069 | 160000 | 8 | 607018 | 5.308 | 0.664 | 0.068 |
| 20000 | 10 | 72707 | 5.537 | 0.554 | 0.085 | 160000 | 10 | 578111 | 5.574 | 0.557 | 0.084 |
| 20000 | 14 | 60222 | 6.685 | 0.478 | 0.081 | 160000 | 14 | 473992 | 6.798 | 0.486 | 0.078 |
| 20000 | 20 | 63555 | 6.335 | 0.317 | 0.110 | 160000 | 20 | 501503 | 6.425 | 0.321 | 0.108 |
| 20000 | 28 | 70979 | 5.672 | 0.203 | 0.142 | 160000 | 28 | 548945 | 5.870 | 0.210 | 0.136 |
| 20000 | 40 | 89909 | 4.478 | 0.112 | 0.199 | 160000 | 40 | 701408 | 4.594 | 0.115 | 0.193 |

**Table 27.3** AntiprotonSeq/AntiprotonClu running-time metrics with $s$ = 5,000 and $n$ varying

| *n* | *K* | *T* | *Spdup* | *Eff* | *EDSF* | *n* | *K* | *T* | *Spdup* | *Eff* | *EDSF* |
|---:|---:|---:|---:|---:|---:|---:|---:|---:|---:|---:|---:|
| 1000 | seq | 100685 | | | | 2800 | seq | 801269 | | | |
| 1000 | 1 | 102854 | 0.979 | 0.979 | | 2800 | 1 | 809229 | 0.990 | 0.990 | |
| 1000 | 2 | 52639 | 1.913 | 0.956 | 0.024 | 2800 | 2 | 410400 | 1.952 | 0.976 | 0.014 |
| 1000 | 3 | 36744 | 2.740 | 0.913 | 0.036 | 2800 | 3 | 275302 | 2.911 | 0.970 | 0.010 |
| 1000 | 4 | 29036 | 3.468 | 0.867 | 0.043 | 2800 | 4 | 207113 | 3.869 | 0.967 | 0.008 |
| 1000 | 5 | 24858 | 4.050 | 0.810 | 0.052 | 2800 | 5 | 166798 | 4.804 | 0.961 | 0.008 |
| 1000 | 6 | 22186 | 4.538 | 0.756 | 0.059 | 2800 | 6 | 142222 | 5.634 | 0.939 | 0.011 |
| 1000 | 8 | 19517 | 5.159 | 0.645 | 0.074 | 2800 | 8 | 109660 | 7.307 | 0.913 | 0.012 |
| 1000 | 10 | 18575 | 5.420 | 0.542 | 0.090 | 2800 | 10 | 90965 | 8.809 | 0.881 | 0.014 |
| 1000 | 14 | 15604 | 6.453 | 0.461 | 0.086 | 2800 | 14 | 68671 | 11.668 | 0.833 | 0.014 |
| 1000 | 20 | 16535 | 6.089 | 0.304 | 0.117 | 2800 | 20 | 54368 | 14.738 | 0.737 | 0.018 |
| 1000 | 28 | 18828 | 5.348 | 0.191 | 0.153 | 2800 | 28 | 48407 | 16.553 | 0.591 | 0.025 |
| 1000 | 40 | 23815 | 4.228 | 0.106 | 0.212 | 2800 | 40 | 45669 | 17.545 | 0.439 | 0.032 |
| 1400 | seq | 200576 | | | | 4000 | seq | 1634195 | | | |
| 1400 | 1 | 201381 | 0.996 | 0.996 | | 4000 | 1 | 1659920 | 0.985 | 0.985 | |
| 1400 | 2 | 102207 | 1.962 | 0.981 | 0.015 | 4000 | 2 | 844169 | 1.936 | 0.968 | 0.017 |
| 1400 | 3 | 69723 | 2.877 | 0.959 | 0.019 | 4000 | 3 | 565780 | 2.888 | 0.963 | 0.011 |
| 1400 | 4 | 54082 | 3.709 | 0.927 | 0.025 | 4000 | 4 | 425581 | 3.840 | 0.960 | 0.009 |
| 1400 | 5 | 44872 | 4.470 | 0.894 | 0.029 | 4000 | 5 | 342431 | 4.772 | 0.954 | 0.008 |
| 1400 | 6 | 39045 | 5.137 | 0.856 | 0.033 | 4000 | 6 | 288294 | 5.669 | 0.945 | 0.008 |
| 1400 | 8 | 32254 | 6.219 | 0.777 | 0.040 | 4000 | 8 | 220023 | 7.427 | 0.928 | 0.009 |
| 1400 | 10 | 28744 | 6.978 | 0.698 | 0.047 | 4000 | 10 | 179766 | 9.091 | 0.909 | 0.009 |
| 1400 | 14 | 23383 | 8.578 | 0.613 | 0.048 | 4000 | 14 | 131921 | 12.388 | 0.885 | 0.009 |
| 1400 | 20 | 22171 | 9.047 | 0.452 | 0.063 | 4000 | 20 | 99536 | 16.418 | 0.821 | 0.010 |
| 1400 | 28 | 23100 | 8.683 | 0.310 | 0.082 | 4000 | 28 | 81020 | 20.170 | 0.720 | 0.014 |
| 1400 | 40 | 27096 | 7.402 | 0.185 | 0.112 | 4000 | 40 | 70798 | 23.083 | 0.577 | 0.018 |
| 2000 | seq | 408967 | | | | 5600 | seq | 3201878 | | | |
| 2000 | 1 | 410678 | 0.996 | 0.996 | | 5600 | 1 | 3254822 | 0.984 | 0.984 | |
| 2000 | 2 | 207915 | 1.967 | 0.983 | 0.013 | 5600 | 2 | 1621172 | 1.975 | 0.988 | -0.004 |
| 2000 | 3 | 139849 | 2.924 | 0.975 | 0.011 | 5600 | 3 | 1079498 | 2.966 | 0.989 | -0.003 |
| 2000 | 4 | 106687 | 3.833 | 0.958 | 0.013 | 5600 | 4 | 816549 | 3.921 | 0.980 | 0.001 |
| 2000 | 5 | 87143 | 4.693 | 0.939 | 0.015 | 5600 | 5 | 657699 | 4.868 | 0.974 | 0.003 |
| 2000 | 6 | 74657 | 5.478 | 0.913 | 0.018 | 5600 | 6 | 553896 | 5.781 | 0.963 | 0.004 |
| 2000 | 8 | 59171 | 6.912 | 0.864 | 0.022 | 5600 | 8 | 419712 | 7.629 | 0.954 | 0.005 |
| 2000 | 10 | 50759 | 8.057 | 0.806 | 0.026 | 5600 | 10 | 340806 | 9.395 | 0.940 | 0.005 |
| 2000 | 14 | 38993 | 10.488 | 0.749 | 0.025 | 5600 | 14 | 247764 | 12.923 | 0.923 | 0.005 |
| 2000 | 20 | 33737 | 12.122 | 0.606 | 0.034 | 5600 | 20 | 181130 | 17.677 | 0.884 | 0.006 |
| 2000 | 28 | 32054 | 12.759 | 0.456 | 0.044 | 5600 | 28 | 141759 | 22.587 | 0.807 | 0.008 |
| 2000 | 40 | 33138 | 12.341 | 0.309 | 0.057 | 5600 | 40 | 113055 | 28.321 | 0.708 | 0.010 |



CHAPTER 28

# Scalability and Pipelining

in which we characterize a parallel program's ability to scale up to more processors; we consider the memory scalability of the antiproton motion program; we devise a design pattern that improves the memory scalability of cluster parallel programs; and we apply the design pattern

# CHAPTER 28 Scalability and Pipelining

## 28.1 Scalability

In Chapter 27, we developed a cluster parallel program to compute the motion of antiprotons. Let's contemplate running the program on more parallel processors, so that we can calculate the motion of more particles. The term **scalability** refers to the *ability* of a parallel program to calculate larger problems as the number of processors *scales up* (increases). What is the cluster parallel particle motion program's scalability? And are there any limits to its scalability?

One way to approach this question is to see what happens to the problem size $N$ as the number of processors $K$ increases, keeping everything else the same. One metric often cited is the "isoefficiency function," which gives a formula for $N$ as a function of $K$ such that the parallel program's *efficiency,* and hence its speedup, is held the same as $K$ increases. However, as discussed in Chapter 10, a more meaningful metric gives $N$ as a function of $K$ when the *running time* is held the same—namely, the program's **sizeup**. For a fixed running-time budget, the sizeup tells exactly how much larger a problem can be computed with more processors. Thus, sizeup is one measure of scalability.

Another way to approach a program's scalability is to see what happens to the **input size** $n$, rather than the problem size $N$, as the number of processors increases. While the amount of computation is proportional to $N$ by definition, the relationship between the amount of computation and the input size depends on the algorithm's complexity. For the AES key search program, the amount of computation is $O(2^n)$, where $n$ is the number of missing key bits. For Floyd's Algorithm, the amount of computation is $O(n^3)$, where $n$ is the number of vertices in the graph. For the particle motion program, the amount of computation is $O(n^2)$, where $n$ is the number of particles. Thus, we defined $N$ to be $2^n$, $n^3$, and $n^2$, respectively, for these programs. Let $f(n)$ be the function that maps input size to problem size; then the inverse function $f^{-1}(N)$ maps problem size to input size. For the AES key search program, $f^{-1}(N) = \log_2 N$; for Floyd's Algorithm, $f^{-1}(N) = N^{1/3}$; for the particle motion program, $f^{-1}(N) = N^{1/2}$. A general formula for input size $n$ as a function of running time $T$ and number of processors $K$ is the following:

$$n(T, K) = f^{-1}[N(T, K)] \tag{28.1}$$

For example, let's say the particle motion program running for time $T$ can solve a problem size of $N$ on $K$ processors. Suppose we scale up to $2K$ processors. If the program's sizeup is ideal, then the program can solve a problem size of $2N$ in time $T$ on $2K$ processors. This, in turn, means the input size can increase from $n$ to $f^{-1}(2N) = (2N)^{1/2} = 1.414n$. Nonideal sizeups result in smaller input size increases. If the program's sizeup efficiency is 90 percent, for example, the program can solve a problem size of $0.9 \cdot 2N$ in time $T$ on $2K$ processors. This, in turn, means the input size can increase from $n$ to $f^{-1}(0.9 \cdot 2N) = (0.9 \cdot 2N)^{1/2} = 1.342n$.

However, there might be a limit on how far we can scale up, because the program's memory usage in each process might depend on $n$, hence on $N$ and $K$. Consider the cluster parallel particle motion program's memory usage (Figure 28.1). Because each process keeps only one slice of the velocity and acceleration arrays, the amount of memory needed for these arrays is proportional to $n/K$. With an ideal sizeup, $n$ is proportional to $K^{1/2}$. Thus, the amount of memory needed for these arrays is proportional to $K^{-1/2}$. That is, the amount of memory in each process for the velocity and acceleration arrays goes *down* as the number of processors goes up (while keeping the running time the same).

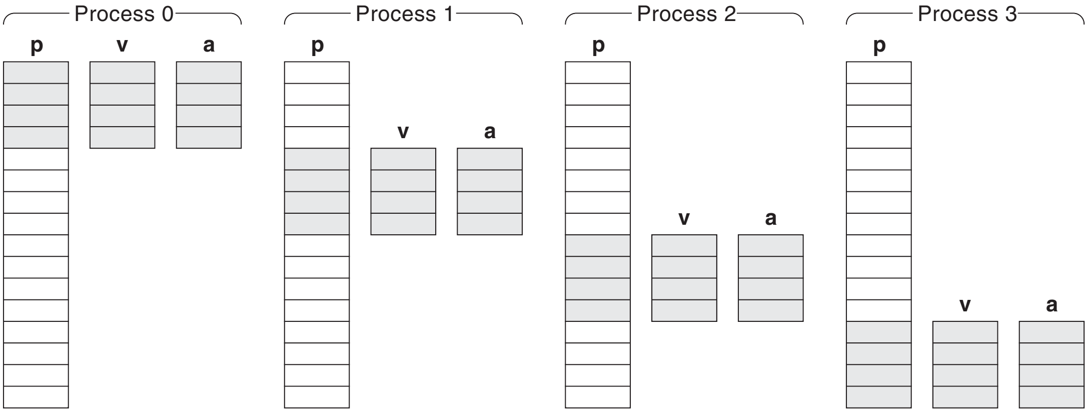


**Figure 28.1** AntiprotonClu program storage allocation in each process

However, each process has to keep the *entire* position array, and the amount of memory needed for this array is proportional to $n$—not $n/K$. Thus, the amount of memory for this array is proportional to $K^{1/2}$. That is, the amount of memory in each process for the position array goes *up* as the number of processors goes up. Eventually, each process consumes all the available memory in each processor, after which we cannot go to a larger problem size by adding more processors (while keeping the running time the same).

When we speak of "memory," we are referring to **physical memory**—the number of bytes in the RAM chips in the processor. A program can use **virtual memory** to access more storage than the physical RAM chips. However, a program whose virtual memory usage exceeds physical memory experiences a severe performance reduction as the operating system swaps virtual memory pages back and forth between RAM and disk. Such a performance reduction is anathema in a parallel program. Never run a parallel program on a problem too large to fit in physical memory.

Let $Mem(n)$ denote the memory needed in each process as a function of the input size. Then by Equation 28.1, the formula

$$Mem(f^{-1}[N(T,K)]) \tag{28.2}$$

gives the memory usage as a function of the number of processors $K$ for a certain running time $T$. If the memory usage is a constant or a decreasing function of $K$, the program will not run out of memory when scaling up (while keeping $T$ the same). If the memory usage is an increasing function of $K$, then there is a limit on how far the program can scale up. Setting Equation 28.2 equal to the available physical memory in each processor and solving for $K$ yields the maximum number of processors that can be utilized.

Not all programs require more memory as the problem size increases. For example, the AES key search program allocates storage for one plaintext, one ciphertext, one trial key, one cipher object, and so on—no matter how many missing key bits there are. Thus, the AES key search program can solve any size problem without hitting a memory limit.

A particle motion program that does not hit a memory limit would be preferable to the program we have now. To achieve this, we need each process to store only one slice of the position array as well as one slice of the velocity and acceleration arrays (Figure 28.2). Then all the arrays will occupy storage proportional to $n/K$, that is, storage proportional to $K^{-1/2}$. We can then scale up the computation to as many particles as we please, and still get the answer in the same time, by increasing the number of processors appropriately, without worrying about running out of memory.

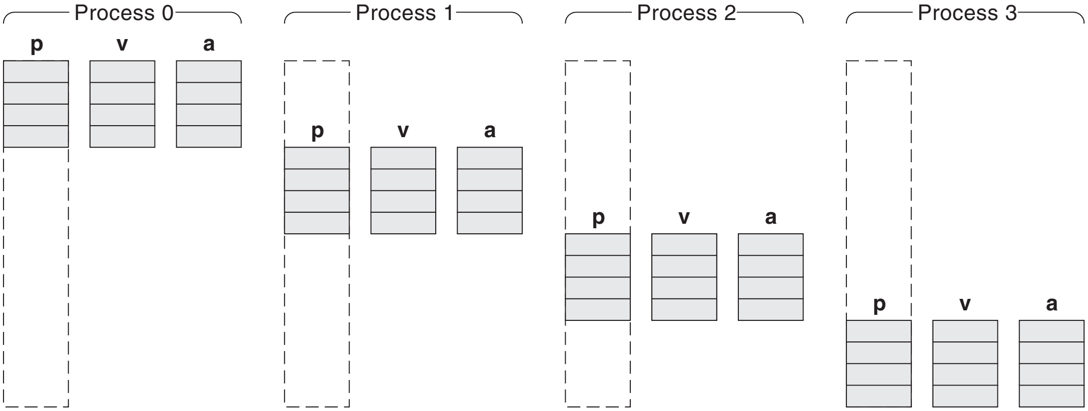


**Figure 28.2** Desired storage allocation in each process

## 28.2 Pipelined Message Passing

To compute the electrostatic forces, each process must pair its own particles' positions with every other process's particles' positions. But if each process stores only its own slice of the position array, the process does not have all the data it needs to compute the electrostatic forces.

To solve this problem, we must rearrange the program's computation and communication. Instead of computing *all* the electrostatic forces followed by communicating *all* the particle positions (with an all-gather), we must *intersperse* computing a *slice* of the electrostatic forces with communicating a *slice* of the particle positions. For example, suppose there are 10,000 particles, and suppose the program is divided among four parallel processes. Here's what each process would do:

1. Compute the forces between all pairs of particles with both particles in the process's own slice, accumulating the forces into the acceleration array slice.

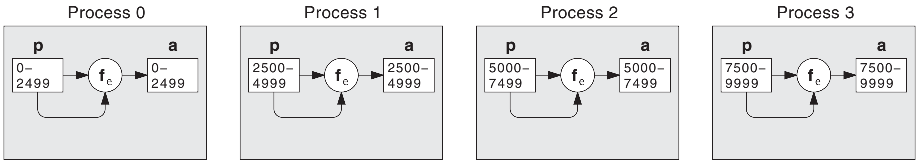


**Figure 28.3** First round of computation

2. Send the process's slice of the position array to the previous process. Simultaneously, receive the next process's slice of the position array into an auxiliary buffer. (There is also another auxiliary buffer that will be used later.)

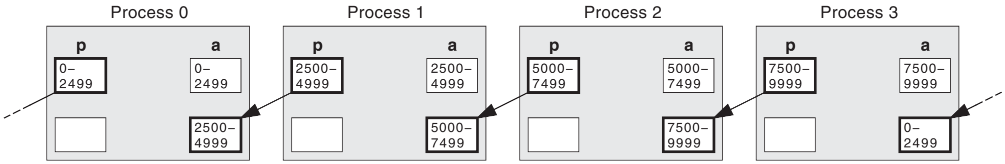


**Figure 28.4** First round of communication

3. Swap the auxiliary buffers, and then compute the forces between all pairs of particles with one particle in the process's own slice and the other particle in the just-received slice (which contains particle positions from the process one rank ahead).

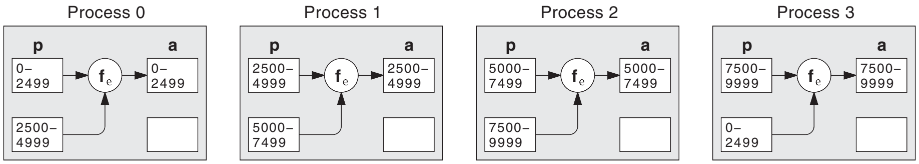


**Figure 28.5** Second round of computation

4. Send the position array slice from the first auxiliary buffer to the previous process. Simultaneously, receive a position array slice into the second auxiliary buffer from the next process.

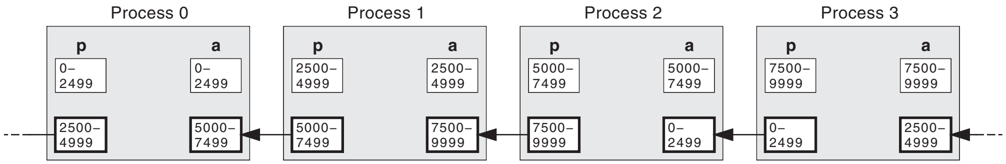


**Figure 28.6** Second round of communication

5. Swap the auxiliary buffers, and then compute the forces between all pairs of particles with one particle in the process's own slice and the other particle in the just-received slice (which contains particle positions from the process two ranks ahead).

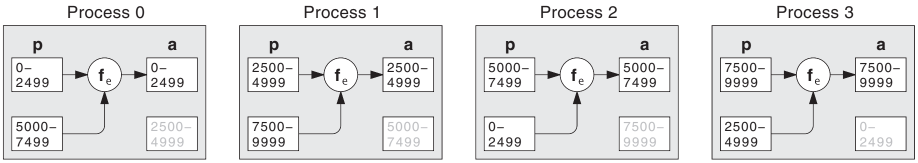


**Figure 28.7** Third round of computation

6. Send the position array slice from the first auxiliary buffer to the previous process. Simultaneously, receive a position array slice into the second auxiliary buffer from the next process.

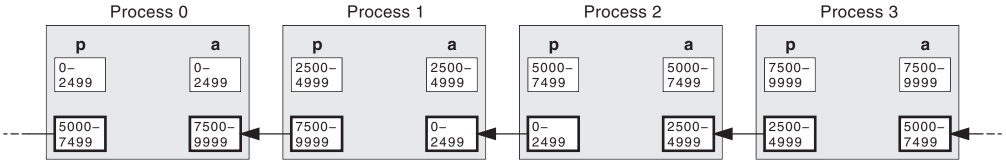


**Figure 28.8** Third round of communication

7. Swap the auxiliary buffers, and then compute the forces between all pairs of particles with one particle in the process's own slice and the other particle in the just-received slice (which contains particle positions from the process three ranks ahead).

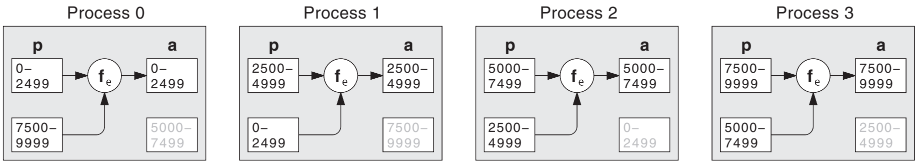


**Figure 28.9** Fourth round of computation

This concludes the calculation of the electrostatic forces. The rest of the time step's computations—calculating the magnetic forces and updating the positions and velocities—are the same as in the AntiprotonClu program.

The new particle motion program follows the **pipelined message passing** pattern. From any individual process's point of view, the other processes' slices of the particle position array arrive in a steady stream, like oil from a pipeline. Each process pumps the slices in, the process does some computation with them, and then it pumps them out again.

## 28.3 Point-to-Point Communication: Send-Receive

At each communication step in the pipelined message passing pattern, each process both sends a message to another process and receives a message from a third process. We could write the program so each process sends the outgoing message first, and then goes back and receives the incoming message. However, if all the processes are sending and none are receiving, and if the network buffers inside each processor's operating system kernel fill up, then flow control in the network transport protocol would halt transmission in all the processes, the sends would never finish, and the program would go into a deadlock. To avoid the deadlock, we could write the program so half the processes send the outgoing message while the other half receive the incoming message, and then switch and communicate the other way round. But this would require some tricky coding.

We can avoid the deadlock and simplify the coding by employing the **send-receive** operation. With a single method call, each process simultaneously sends one message and receives another message.

```
int toRank = (rank – 1 + size) % size;
int fromRank = (rank + 1) % size;
world.sendReceive (toRank, outBuf, fromRank, inBuf);
```

The `sendReceive()` method's first argument is the rank of the process to which to send the outgoing message. In the pipelined particle motion program, this is always the process at the next lower rank. The variable `toRank` is set to this process's rank minus 1, wrapping around to rank `size-1` if this process's rank is 0. The second argument is the buffer from which to obtain the outgoing message data. The third argument is the rank of the process from which to receive the incoming message. In the pipelined particle motion program, this is always the process at the next higher rank. The variable `fromRank` is set to this process's rank plus 1, wrapping around to rank 0 if this process's rank is `size-1`. The fourth argument is the buffer in which to store the incoming message data. The incoming data must be stored in a different place from the outgoing data, so as not to wipe out the outgoing data before it can be sent.

Under the hood, the `sendReceive()` method sends the outgoing message and receives the incoming message simultaneously in separate threads. If the network interface supports full duplex operation, both the outgoing and the incoming messages can traverse the network at the same time, resulting in a smaller communication time than if the messages had to travel sequentially. Because the receive is happening at the same time as the send, a deadlock will not occur (as it might if the receive happened after the send).

How much time will the pipelined particle motion program's message passing take? With *K* processes, there are *K*–1 rounds of message passing during each time step. During one round, each process simultaneously sends one message and receives one message. Each message consists of one slice of the particle position array, or *n*/*K* position vectors. These are exactly the same messages the original program's all-gather operation does all at once. Thus, the pipelined program's communication time is the same as the original program's. Because the pipelined program does the same computations, its calculation time is also the same as the original program's. Thus, the pipelined program should exhibit the same performance as the original program, including the slowdown as *K* increases.

We went to a pipelined design not to reduce the program's running time, but to improve the program's memory scalability. In the pipelined program, all the arrays—including the two auxiliary buffers used for communication—have *n*/*K* elements.

## 28.4 Pipelined Antiproton Program

Here is the source code for class edu.rit.clu.antimatter.AntiprotonClu2, the cluster parallel particle motion program with pipelined message passing.

```
package edu.rit.clu.antimatter;
import edu.rit.io.Files;
import edu.rit.mp.DoubleBuf;
import edu.rit.pj.Comm;
import edu.rit.util.Random;
import edu.rit.util.Range;
import edu.rit.vector.Vector2D;
import java.io.BufferedOutputStream;
import java.io.File;
import java.io.FileOutputStream;
public class AntiprotonClu2
   {
   // Charge on an antiproton.
   static final double QP = 3.0;

   // Magnetic field strength.
   static final double B = 3.0;

   static final double QP_QP = QP * QP;
   static final double QP_B = QP * B;
   // World communicator.
   static Comm world;
   static int size;
   static int rank;
   static int predRank;
   static int succRank;

   // Command line arguments.
   static long seed;
   static double R;
   static double dt;
   static int steps;
   static int snaps;
   static int N;
   static File outfile;

   static double one_half_dt_sqr;

   // Antiproton slices.
   static Range[] slices;
   static Range mySlice;
   static int myLb;
   static int myLen;

   // Acceleration, velocity, and position vector arrays.
   static Vector2D[] a;
   static Vector2D[] v;
   static Vector2D[] p;

   // Position vector arrays to use for pipelined message passing.
   static Vector2D[] p2;
   static Vector2D[] p3;

   // Position vector array communication buffers.
   static DoubleBuf pbuf;
   static DoubleBuf p2buf;
   static DoubleBuf p3buf;

   // Temporary storage.
   static Vector2D temp = new Vector2D();

   // Total momentum.
   static Vector2D totalMV = new Vector2D();

   /**
    * Main program.
    */
   public static void main
      (String[] args)
      throws Exception
      {
      // Start timing.
      long t1 = System.currentTimeMillis();

      // Initialize world communicator.
      Comm.init (args);
      world = Comm.world();
      size = world.size();
      rank = world.rank();
```

Here, we determine the ranks of the two processes with which this process will be communicating.

```
      predRank = (rank - 1 + size) % size;
      succRank = (rank + 1) % size;

      // Parse command line arguments.
      if (args.length != 7) usage();
      seed = Long.parseLong (args[0]);
      R = Double.parseDouble (args[1]);
      dt = Double.parseDouble (args[2]);
      steps = Integer.parseInt (args[3]);
      snaps = Integer.parseInt (args[4]);
      N = Integer.parseInt (args[5]);
      outfile = new File (args[6]);

      one_half_dt_sqr = 0.5 * dt * dt;
```

Each process is aware of the index range for every process's slice of the acceleration, velocity, and position vector arrays.

```
      // Set up antiproton slices.
      slices = new Range (0, N-1) .subranges (size);
      mySlice = slices[rank];
      myLb = mySlice.lb();
      myLen = mySlice.length();
```

Because this process is going to initialize only its own slice of the position vector array, this process must skip the pseudorandom number generator over the preceding processes' random numbers. This is the *sequence splitting* pattern.

```
      // Create pseudorandom number generator.
      Random prng = Random.getInstance (seed);
      prng.skip (2 * myLb);

      // Initialize acceleration, velocity, and position vector
      // arrays with this process's slice of antiprotons.
      a = new Vector2D [myLen];
      v = new Vector2D [myLen];
      p = new Vector2D [myLen];
      for (int i = 0; i < myLen; ++ i)
         {
         a[i] = new Vector2D();
         v[i] = new Vector2D();
         p[i] = new Vector2D
            (prng.nextDouble()*R/2+R/4, prng.nextDouble()*R/2+R/4);
         }
```

The auxiliary arrays `p2` and `p3` are used to hold the outgoing and incoming slices of the position vector array, respectively. One minor detail is that these arrays are allocated to hold the same number of elements as this process's slice, plus one. This is necessary because another process's slice might be one larger than this process's slice, if the number of particles is not evenly divisible by the number of processes.

```
      // Initialize position vector arrays for pipelined message
      // passing.
      p2 = new Vector2D [myLen+1];
      p3 = new Vector2D [myLen+1];
      for (int i = 0; i <= myLen; ++ i)
         {
         p2[i] = new Vector2D();
         p3[i] = new Vector2D();
         }

      // Set up position array communication buffers.
      pbuf = Vector2D.doubleBuffer (p);
      p2buf = Vector2D.doubleBuffer (p2);
      p3buf = Vector2D.doubleBuffer (p3);

      // Set up output file and write initial snapshot.
      AntiprotonFile out =
         new AntiprotonFile
            (seed, R, dt, steps, snaps+1, N, myLb, myLen);
      AntiprotonFile.Writer writer =
         out.prepareToWrite
```

```
            (new BufferedOutputStream
               (new FileOutputStream
                  (Files.fileForRank (outfile, rank))));
      writer.writeSnapshot (p, 0, totalMV);

      long t2 = System.currentTimeMillis();

      // Do <snaps> snapshots.
      for (int s = 0; s < snaps; ++ s)
         {
         // Advance time by <steps> steps.
         for (int t = 0; t < steps; ++ t)
            {
```

Here is the first round of computation within one time step (Figure 28.3).

```
            // Compute accelerations due to this process's
            // antiprotons.
            computeAccelerationThisSlice();

            // Do <size>-1 rounds of pipelined message passing.
            DoubleBuf outbuf = pbuf;
            DoubleBuf inbuf = p3buf;
            for (int k = 1; k < size; ++ k)
               {
```

Here is a round of communication within one time step. The first time, `outbuf` refers to this process's slice of particle positions, `p` (Figure 28.4). After that, outbuf refers to the auxiliary array `p2` (Figures 28.6 and 28.8). `inbuf` always refers to the auxiliary array `p3`.

```
               // Shift position slices through the pipeline.
               world.sendReceive
                  (predRank, outbuf, succRank, inbuf);

               // Swap outgoing and incoming position slices and
               // buffers.
               Vector2D[] ptmp = p2;
               p2 = p3;
               p3 = ptmp;
               DoubleBuf tmpbuf = p2buf;
               p2buf = p3buf;
```

```
            p3buf = tmpbuf;
            outbuf = p2buf;
            inbuf = p3buf;
```

Here are the second and subsequent rounds of computation within one time step (Figures 28.5, 28.7, and 28.9). Note that the other process's rank must be specified when calling `computeAccelerationOtherSlice()`; this is needed to determine the length of the other process's slice. The other process's particle positions are located in the auxiliary array `p2`.

```
            // Compute accelerations due to other process's
            // antiprotons.
            computeAccelerationOtherSlice ((rank + k) % size);
            }

         // Move this process's antiprotons.
         step();
         }

      // Compute total momentum.
      computeTotalMomentum();

      // Write snapshot.
      writer.writeSnapshot (p, 0, totalMV);
      }

   // Close output file.
   writer.close();

   // Stop timing.
   long t3 = System.currentTimeMillis();
   System.out.println ((t2-t1) + " msec pre " + rank);
   System.out.println ((t3-t2) + " msec calc " + rank);
   System.out.println ((t3-t1) + " msec total " + rank);
   }

/**
 * Compute this process's slice of the antiproton accelerations
 * due to the repulsive forces from this process's slice of the
 * antiprotons.
 */
private static void computeAccelerationThisSlice()
   {
   // Accumulate forces between each pair of antiprotons, but
   // not between an antiproton and itself.
```

```
      for (int i = 0; i < myLen; ++ i)
         {
         Vector2D a_i = a[i];
         Vector2D p_i = p[i];
         for (int j = 0; j < i; ++ j)
            {
            temp.assign (p_i);
            temp.sub (p[j]);
            double dsqr = temp.sqrMag();
            temp.mul (QP_QP / (dsqr * Math.sqrt(dsqr)));
            a_i.add (temp);
            }

         for (int j = i+1; j < myLen; ++ j)
            {
            temp.assign (p_i);
            temp.sub (p[j]);
            double dsqr = temp.sqrMag();
            temp.mul (QP_QP / (dsqr * Math.sqrt(dsqr)));
            a_i.add (temp);
            }
         }
      }

   /**
    * Compute this process's slice of the antiproton accelerations
    * due to the repulsive forces from another process's slice of
    * the antiprotons (located in p2).
    *
    * @param  fromRank  Other process's rank.
    */
   private static void computeAccelerationOtherSlice
      (int fromRank)
      {
      int otherLen = slices[fromRank].length();

      // Accumulate forces between each pair of antiprotons.
      for (int i = 0; i < myLen; ++ i)
         {
         Vector2D a_i = a[i];
         Vector2D p_i = p[i];
         for (int j = 0; j < otherLen; ++ j)
            {
            temp.assign (p_i);
            temp.sub (p2[j]);
```

```
             double dsqr = temp.sqrMag();
            temp.mul (QP_QP / (dsqr * Math.sqrt(dsqr)));
            a_i.add (temp);
            }
         }
      }

   /**
    * Take one time step.
    */
   private static void step()
      {
      // Move all antiprotons in this slice.
      for (int i = 0; i < myLen; ++ i)
         {
         Vector2D a_i = a[i];
         Vector2D v_i = v[i];
         Vector2D p_i = p[i];

         // Accumulate acceleration on antiproton from magnetic
         // field.
         temp.assign (v_i) .mul (QP_B) .rotate270();
         a_i.add (temp);

         // Update antiproton’s position and velocity.
         temp.assign (v_i);
         p_i.add (temp.mul (dt));
         temp.assign (a_i);
         p_i.add (temp.mul (one_half_dt_sqr));
         temp.assign (a_i);
         v_i.add (temp.mul (dt));

         // Clear antiproton’s acceleration for the next step.
         a_i.clear();
         }
      }

   /**
    * Compute the total momentum for this process’s slice of the
    * antiprotons. The answer is stored in totalMV.
    */
   private static void computeTotalMomentum()
      {
      totalMV.clear();
      for (int i = 0; i < myLen; ++ i)
```

```
         {
         totalMV.add (v[i]);
         }
      }
   }
```

## 28.5 Pipelined Program Performance

Table 28.1 (at the end of the chapter) lists, and Figure 28.10 plots, the AntiprotonClu2 program's performance on the "tardis" parallel computer. The program was run with the following command,

```
$ java -Dpj.np=$K edu.rit.clu.antimatter.AntiprotonClu2 \
  142857 7 0.00001 1000 5 $N outfile.dat
```

where the number of particles ($N) was 1,000, 1,400, 2,000, 2,800, 4,000, or 5,600, the number of time steps between snapshots was 1,000, and the number of snapshots was 5, for a total of 5,000 time steps. Comparing Table 28.1 to Table 27.3, the pipelined program's running times are about the same as the nonpipelined program's running times, as expected.

While the pipelined message passing improved the particle motion program's memory scalability, it did not alter the program's performance. However, the pipelined message passing does make possible another design change that *will* improve the program's performance. This will be the subject of the next chapter.

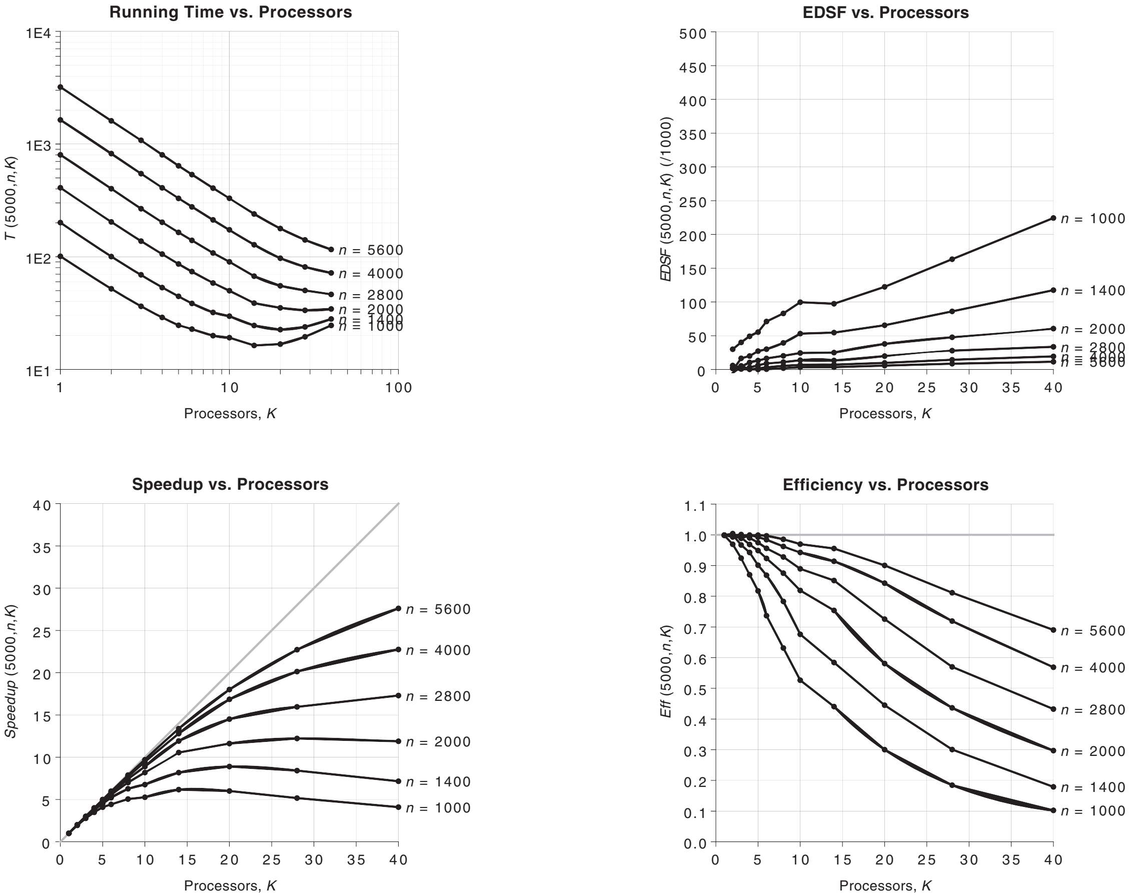
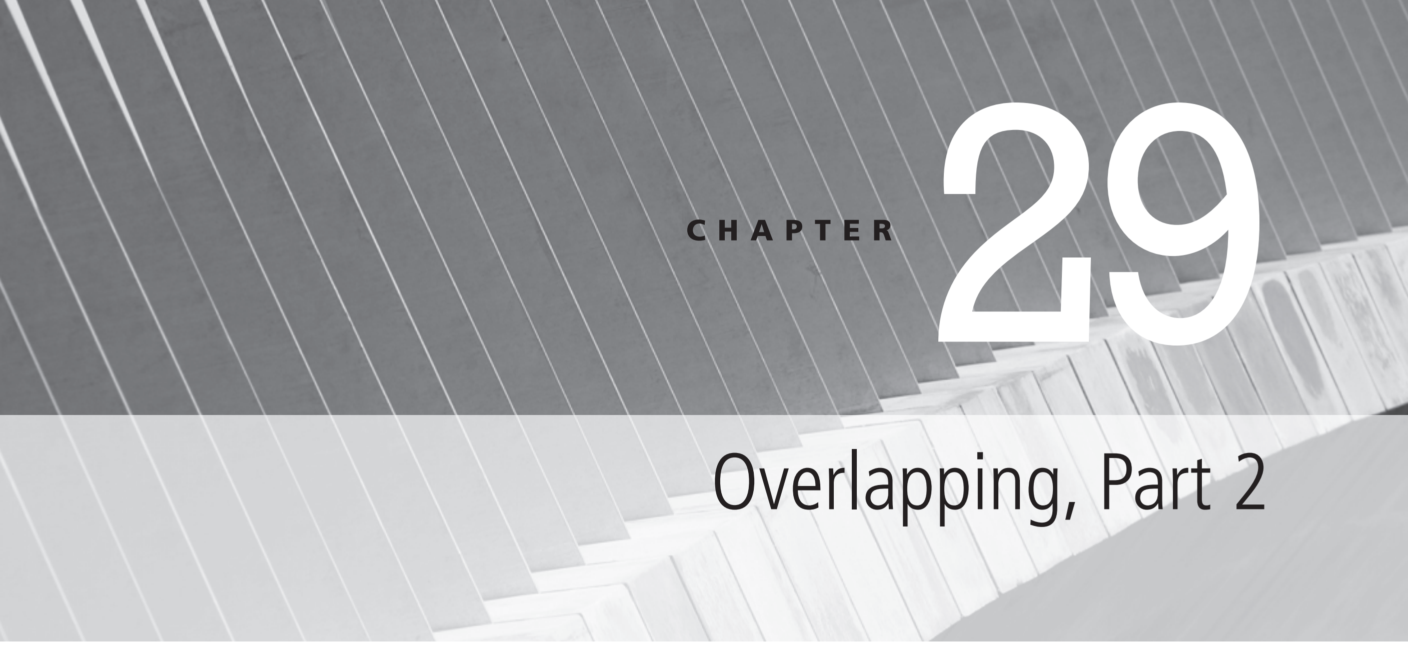


**Figure 28.10** AntiprotonSeq/AntiprotonClu2 running-time metrics with $s$ = 5,000 and $n$ varying

**Table 28.1** AntiprotonSeq/AntiprotonClu2 running-time metrics with $s$ = 5000 and $n$ varying

| *n* | *K* | *T* | *Spdup* | *Eff* | *EDSF* | *n* | *K* | *T* | *Spdup* | *Eff* | *EDSF* |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1000 | seq | 100692 | | | | 2800 | seq | 801121 | | | |
| 1000 | 1 | 100861 | 0.998 | 0.998 | | 2800 | 1 | 801882 | 0.999 | 0.999 | |
| 1000 | 2 | 51944 | 1.938 | 0.969 | 0.030 | 2800 | 2 | 401528 | 1.995 | 0.998 | 0.001 |
| 1000 | 3 | 36321 | 2.772 | 0.924 | 0.040 | 2800 | 3 | 266748 | 3.003 | 1.001 | -0.001 |
| 1000 | 4 | 28935 | 3.480 | 0.870 | 0.049 | 2800 | 4 | 202300 | 3.960 | 0.990 | 0.003 |
| 1000 | 5 | 24636 | 4.087 | 0.817 | 0.055 | 2800 | 5 | 164384 | 4.873 | 0.975 | 0.006 |
| 1000 | 6 | 22783 | 4.420 | 0.737 | 0.071 | 2800 | 6 | 139652 | 5.737 | 0.956 | 0.009 |
| 1000 | 8 | 19935 | 5.051 | 0.631 | 0.083 | 2800 | 8 | 107908 | 7.424 | 0.928 | 0.011 |
| 1000 | 10 | 19130 | 5.264 | 0.526 | 0.100 | 2800 | 10 | 90082 | 8.893 | 0.889 | 0.014 |
| 1000 | 14 | 16324 | 6.168 | 0.441 | 0.097 | 2800 | 14 | 67222 | 11.918 | 0.851 | 0.013 |
| 1000 | 20 | 16772 | 6.004 | 0.300 | 0.122 | 2800 | 20 | 55216 | 14.509 | 0.725 | 0.020 |
| 1000 | 28 | 19489 | 5.167 | 0.185 | 0.163 | 2800 | 28 | 50207 | 15.956 | 0.570 | 0.028 |
| 1000 | 40 | 24584 | 4.096 | 0.102 | 0.224 | 2800 | 40 | 46316 | 17.297 | 0.432 | 0.034 |
| 1400 | seq | 200671 | | | | 4000 | seq | 1634275 | | | |
| 1400 | 1 | 200862 | 0.999 | 0.999 | | 4000 | 1 | 1635910 | 0.999 | 0.999 | |
| 1400 | 2 | 100317 | 2.000 | 1.000 | -0.001 | 4000 | 2 | 822769 | 1.986 | 0.993 | 0.006 |
| 1400 | 3 | 69172 | 2.901 | 0.967 | 0.017 | 4000 | 3 | 546088 | 2.993 | 0.998 | 0.001 |
| 1400 | 4 | 53220 | 3.771 | 0.943 | 0.020 | 4000 | 4 | 408822 | 3.998 | 0.999 | 0.000 |
| 1400 | 5 | 44543 | 4.505 | 0.901 | 0.027 | 4000 | 5 | 329310 | 4.963 | 0.993 | 0.002 |
| 1400 | 6 | 38514 | 5.210 | 0.868 | 0.030 | 4000 | 6 | 276772 | 5.905 | 0.984 | 0.003 |
| 1400 | 8 | 32032 | 6.265 | 0.783 | 0.039 | 4000 | 8 | 212301 | 7.698 | 0.962 | 0.005 |
| 1400 | 10 | 29689 | 6.759 | 0.676 | 0.053 | 4000 | 10 | 173365 | 9.427 | 0.943 | 0.007 |
| 1400 | 14 | 24535 | 8.179 | 0.584 | 0.055 | 4000 | 14 | 127725 | 12.795 | 0.914 | 0.007 |
| 1400 | 20 | 22547 | 8.900 | 0.445 | 0.066 | 4000 | 20 | 96985 | 16.851 | 0.843 | 0.010 |
| 1400 | 28 | 23842 | 8.417 | 0.301 | 0.086 | 4000 | 28 | 81167 | 20.135 | 0.719 | 0.014 |
| 1400 | 40 | 28033 | 7.158 | 0.179 | 0.118 | 4000 | 40 | 71845 | 22.747 | 0.569 | 0.019 |
| 2000 | seq | 408964 | | | | 5600 | seq | 3202221 | | | |
| 2000 | 1 | 409399 | 0.999 | 0.999 | | 5600 | 1 | 3204736 | 0.999 | 0.999 | |
| 2000 | 2 | 203655 | 2.008 | 1.004 | -0.005 | 5600 | 2 | 1606363 | 1.993 | 0.997 | 0.002 |
| 2000 | 3 | 137529 | 2.974 | 0.991 | 0.004 | 5600 | 3 | 1079674 | 2.966 | 0.989 | 0.005 |
| 2000 | 4 | 105541 | 3.875 | 0.969 | 0.010 | 5600 | 4 | 801837 | 3.994 | 0.998 | 0.000 |
| 2000 | 5 | 86207 | 4.744 | 0.949 | 0.013 | 5600 | 5 | 641054 | 4.995 | 0.999 | 0.000 |
| 2000 | 6 | 73826 | 5.540 | 0.923 | 0.016 | 5600 | 6 | 535393 | 5.981 | 0.997 | 0.000 |
| 2000 | 8 | 58415 | 7.001 | 0.875 | 0.020 | 5600 | 8 | 406158 | 7.884 | 0.986 | 0.002 |
| 2000 | 10 | 49946 | 8.188 | 0.819 | 0.024 | 5600 | 10 | 330193 | 9.698 | 0.970 | 0.003 |
| 2000 | 14 | 38749 | 10.554 | 0.754 | 0.025 | 5600 | 14 | 239381 | 13.377 | 0.956 | 0.004 |
| 2000 | 20 | 35202 | 11.618 | 0.581 | 0.038 | 5600 | 20 | 177816 | 18.009 | 0.900 | 0.006 |
| 2000 | 28 | 33470 | 12.219 | 0.436 | 0.048 | 5600 | 28 | 140946 | 22.719 | 0.811 | 0.009 |
| 2000 | 40 | 34403 | 11.887 | 0.297 | 0.061 | 5600 | 40 | 116001 | 27.605 | 0.690 | 0.011 |



# CHAPTER 29

# Overlapping, Part 2

in which we combine two design patterns, pipelining and overlapping, to improve both the antiproton motion program's memory scalability and its performance

# CHAPTER 29 Overlapping, Part 2

## 29.1 Overlapped Computation and Communication

The pipelined antiproton motion program in Chapter 28 improved the memory scalability by storing just one slice of the position vector array in each process. Consequently, the program had to intersperse computation with communication (Figure 29.1).

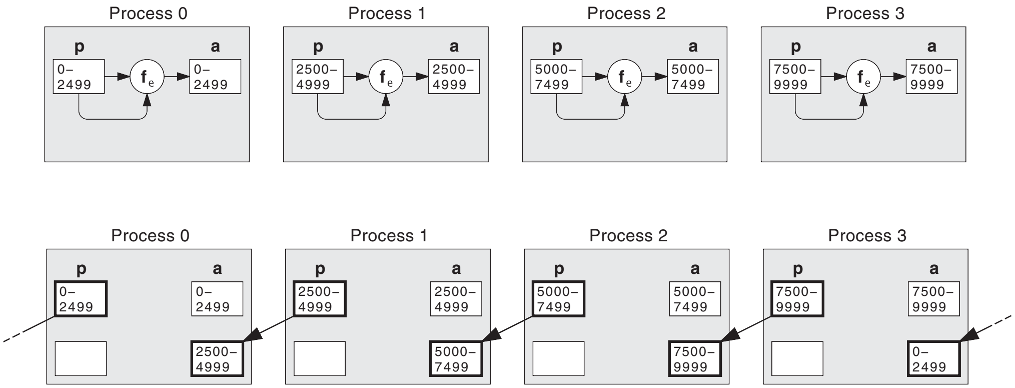


**Figure 29.1** First round of computation followed by communication

During the communication phase, the CPU initiates the send-receive operation, and then sits around twiddling its transistors until the outgoing and incoming messages have finished traveling across the comparatively slow network. That is a waste of CPU time. It would be better if the CPU were doing useful work while the messages traverse the network.

Instead of doing a round of communication *after* doing a round of computation, suppose we do the computation and the communication *at the same time.* We saw a similar pattern in Chapter 18, where we did *overlapped computation and I/O* in an SMP parallel program—computation of the cellular automaton's state, output of the CA image to a file. Now, in the cluster parallel antiproton motion program, we will do **overlapped computation and communication**. For example, suppose there are 10,000 antiprotons, and suppose the program is divided among four parallel processes. Here's what each process would do:

1. Compute the forces between all pairs of antiprotons with both antiprotons in the process's own slice, accumulating the forces into the acceleration array slice; send the process's slice of the position array to the previous process; and receive the next process's slice of the position array into an auxiliary buffer. (There is also another auxiliary buffer that will be used later.)

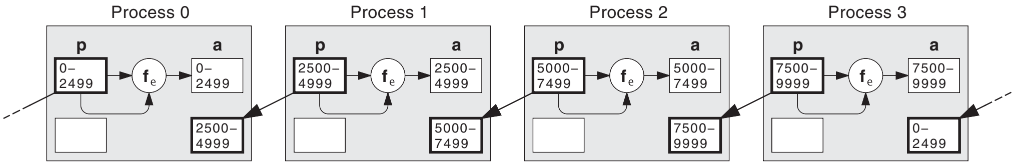


**Figure 29.2** First round of overlapped computation and communication

2. Swap the auxiliary buffers. Then, compute the forces between all pairs of antiprotons with one antiproton in the process's own slice and the other antiproton in the just-received slice (which contains antiproton positions from the process one rank ahead); send the position array slice from the first auxiliary buffer to the previous process; and receive a position array slice into the second auxiliary buffer from the next process.

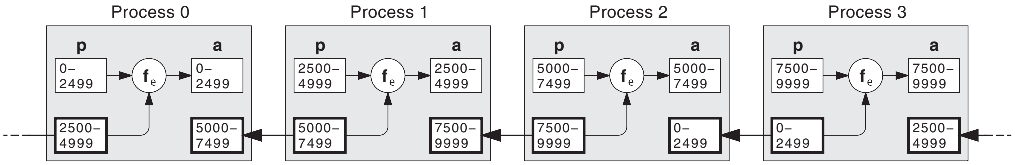


**Figure 29.3** Second round of overlapped computation and communication

3. Swap the auxiliary buffers. Then, compute the forces between all pairs of antiprotons with one antiproton in the process's own slice and the other antiproton in the just-received slice (which contains antiproton positions from the process two ranks ahead); send the position array slice from the first auxiliary buffer to the previous process; and receive a position array slice into the second auxiliary buffer from the next process.

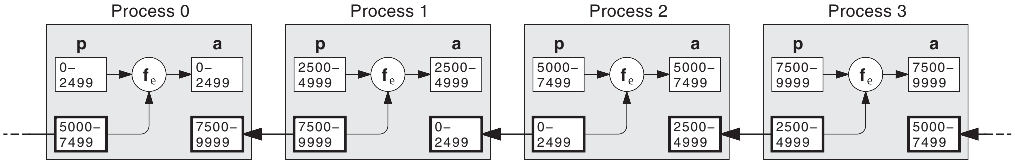


**Figure 29.4** Third round of overlapped computation and communication

4. Swap the auxiliary buffers. Then, compute the forces between all pairs of antiprotons with one antiproton in the process's own slice and the other antiproton in the just-received slice (which contains antiproton positions from the process three ranks ahead). The final round does only computation, not communication.

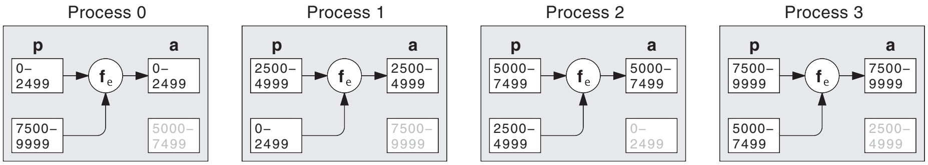


**Figure 29.5** Fourth round of computation

## 29.2 Non-Blocking Send-Receive

To implement the overlapped computation and communication, we will employ a **non-blocking send-receive** operation.

```
int toRank = (rank – 1 + size) % size;
int fromRank = (rank + 1) % size;
CommRequest request = new CommRequest();
world.sendReceive (toRank, outBuf, fromRank, inBuf, request);
// Perform computation
req.waitForFinish();
```

This is the same `sendReceive()` method call as in Chapter 28, except it has an additional argument, a communication request object `(request)`, which is an instance of class edu.rit.pj.CommRequest. The `sendReceive()` method initiates the communication and returns immediately, without waiting for the communication to finish. The main program thread then performs the computations. Under the hood, separate threads send and receive the outgoing and incoming messages and block waiting for the messages to finish. When the main program thread has finished the round of computations, the main program thread must not proceed until the messages have also finished. The main program thread calls the communication request object's `waitForFinish()` method; this call blocks until the outgoing and incoming messages have finished. When the `waitForFinish()` method returns, the main program thread goes on to the next round.

## 29.3 Pipelined Overlapped Antiproton Program

Here is the source code for class edu.rit.clu.antimatter.AntiprotonClu3, the cluster parallel antiproton motion program with pipelined message passing and overlapped computation and communication.

```
package edu.rit.clu.antimatter;
import edu.rit.io.Files;
import edu.rit.mp.DoubleBuf;
import edu.rit.pj.Comm;
import edu.rit.pj.CommRequest;
import edu.rit.util.Random;
import edu.rit.util.Range;
import edu.rit.vector.Vector2D;
import java.io.BufferedOutputStream;
import java.io.File;
import java.io.FileOutputStream;
public class AntiprotonClu3
   {
   // Charge on an antiproton.
   static final double QP = 3.0;

   // Magnetic field strength.
   static final double B = 3.0;

   static final double QP_QP = QP * QP;
   static final double QP_B = QP * B;

   // World communicator.
   static Comm world;
   static int size;
   static int rank;
   static int predRank;
   static int succRank;

   // Command line arguments.
   static long seed;
   static double R;
   static double dt;
   static int steps;
   static int snaps;
   static int N;
   static File outfile;

   static double one_half_dt_sqr;

   // Antiproton slices.
   static Range[] slices;
   static Range mySlice;
   static int myLb;
   static int myLen;
   // Acceleration, velocity, and position vector arrays.
   static Vector2D[] a;
   static Vector2D[] v;
   static Vector2D[] p;

   // Position vector arrays to use for pipelined message passing.
   static Vector2D[] p2;
   static Vector2D[] p3;

   // Position vector array communication buffers.
   static DoubleBuf pbuf;
   static DoubleBuf p2buf;
   static DoubleBuf p3buf;
   static CommRequest request;

   // Temporary storage.
   static Vector2D temp = new Vector2D();

   // Total momentum.
   static Vector2D totalMV = new Vector2D();

   /**
    * Main program.
    */
   public static void main
      (String[] args)
      throws Exception
      {
      // Start timing.
      long t1 = System.currentTimeMillis();

      // Initialize world communicator.
      Comm.init (args);
      world = Comm.world();
      size = world.size();
      rank = world.rank();
      predRank = (rank - 1 + size) % size;
      succRank = (rank + 1) % size;

      // Parse command line arguments.
      if (args.length != 7) usage();
      seed = Long.parseLong (args[0]);
      R = Double.parseDouble (args[1]);
      dt = Double.parseDouble (args[2]);
      steps = Integer.parseInt (args[3]);
      snaps = Integer.parseInt (args[4]);
```

```
N = Integer.parseInt (args[5]);
outfile = new File (args[6]);

one_half_dt_sqr = 0.5 * dt * dt;

// Set up antiproton slices.
slices = new Range (0, N-1) .subranges (size);
mySlice = slices[rank];
myLb = mySlice.lb();
myLen = mySlice.length();

// Create pseudorandom number generator.
Random prng = Random.getInstance (seed);
prng.skip (2 * myLb);

// Initialize acceleration, velocity, and position vector
// arrays with this process's slice of antiprotons.
a = new Vector2D [myLen];
v = new Vector2D [myLen];
p = new Vector2D [myLen];
for (int i = 0; i < myLen; ++ i)
   {
   a[i] = new Vector2D();
   v[i] = new Vector2D();
   p[i] = new Vector2D
      (prng.nextDouble()*R/2+R/4, prng.nextDouble()*R/2+R/4);
   }

// Initialize position vector arrays for pipelined message
// passing.
p2 = new Vector2D [myLen+1];
p3 = new Vector2D [myLen+1];
for (int i = 0; i <= myLen; ++ i)
   {
   p2[i] = new Vector2D();
   p3[i] = new Vector2D();
   }

// Set up position array communication buffers.
pbuf = Vector2D.doubleBuffer (p);
p2buf = Vector2D.doubleBuffer (p2);
p3buf = Vector2D.doubleBuffer (p3);
request = new CommRequest();
```

```
// Set up output file and write initial snapshot.
AntiprotonFile out =
   new AntiprotonFile
      (seed, R, dt, steps, snaps+1, N, myLb, myLen);
AntiprotonFile.Writer writer =
   out.prepareToWrite
      (new BufferedOutputStream
         (new FileOutputStream
            (Files.fileForRank (outfile, rank))));
writer.writeSnapshot (p, 0, totalMV);

long t2 = System.currentTimeMillis();

// Do <snaps> snapshots.
for (int s = 0; s < snaps; ++ s)
   {
   // Advance time by <steps> steps.
   for (int t = 0; t < steps; ++ t)
      {
```

This initiates the first round of overlapped communication, except if there is only one process, then there is no communication.

```
      // Initiate first round of pipelined message passing if
      // any.
      DoubleBuf outbuf = pbuf;
      DoubleBuf inbuf = p3buf;
      if (size > 1)
         {
         world.sendReceive
            (predRank, outbuf, succRank, inbuf, request);
         }
```

Here is the first round of overlapped computation.

```
      // Compute accelerations due to this process's
      // antiprotons, overlapped with communication.
      computeAccelerationThisSlice();

      // Do <size>-1 rounds of pipelined message passing.
      for (int k = 1; k < size; ++ k)
         {
```

Here is the end of the previous round of overlapped computation and communication.

```
                  // Wait for current round to finish.
                  request.waitForFinish();

                  // Swap outgoing and incoming position slices and
                  // buffers.
                  Vector2D[] ptmp = p2;
                  p2 = p3;
                  p3 = ptmp;
                  DoubleBuf tmpbuf = p2buf;
                  p2buf = p3buf;
                  p3buf = tmpbuf;
                  outbuf = p2buf;
                  inbuf = p3buf;
```

This initiates the next round of overlapped communication, except there is no communication during the final round.

```
                  // Initiate next round if any.
                  if (k < size-1)
                     {
                     world.sendReceive
                        (predRank, outbuf, succRank, inbuf, request);
                     }
```

Here is the next round of overlapped computation.

```
                  // Compute accelerations due to other process's
                  // antiprotons, overlapped with communication.
                  computeAccelerationOtherSlice ((rank + k) % size);
                  }

               // Move this process's antiprotons.
               step();
               }

            // Compute total momentum.
            computeTotalMomentum();

            // Write snapshot.
            writer.writeSnapshot (p, 0, totalMV);
            }
```

```
         // Close output file.
         writer.close();

         // Stop timing.
         long t3 = System.currentTimeMillis();
         System.out.println ((t2-t1) + " msec pre " + rank);
         System.out.println ((t3-t2) + " msec calc " + rank);
         System.out.println ((t3-t1) + " msec total " + rank);
         }
      }
```

The subroutines for computing the accelerations due to electrostatic forces, updating the positions and velocities, and calculating the total momentum are the same as in the AntiprotonClu2 program in Chapter 28.

## 29.4 Computation Time Model

Before measuring the pipelined overlapped antiproton motion program's performance, let's derive a model for the program's running time. The AntiprotonClu3 program's calculation time is the same as the AntiprotonClu program in Chapter 27,

$$T_{\text{calc}}(s,n,K) = 2.04\times 10^{-8}\, sn^2\big/K \tag{29.1}$$

where $s$ is the number of time steps, $n$ is the number of antiprotons, $K$ is the number of parallel processors, and $T_{\text{calc}}$ is the calculation time in seconds. Because the AntiprotonClu3 program sends the same messages as the AntiprotonClu and AntiprotonClu2 programs, the AntiprotonClu3 program's communication time is also the same:

$$T_{\text{comm}}(s,n,K) = s(2.08\times 10^{-4} + 1.37\times 10^{-7}\, n\big/K)\ (K-1) \tag{29.2}$$

But because the computation and communication are overlapped, the total running time is the larger of $T_{\text{calc}}$ and $T_{\text{comm}}$, and not the sum:

$$\begin{aligned} T(s,n,K) &= \max[2.04\times 10^{-8}\, sn^2\big/K\,, \\ & s(2.08\times 10^{-4} + 1.37\times 10^{-7}\, n\big/K)\ (K-1)] \end{aligned} \tag{29.3}$$

Figure 29.6 plots the calculation time, communication time, and total computation time as a function of $K$ for $s = 5{,}000$ and $n = 2{,}000$. Figure 29.7 plots the speedup as a function of $K$. The smallest running time, hence the largest speedup, occurs when $T_{\text{calc}}$ equals $T_{\text{comm}}$. Setting Equation 29.1 equal to Equation 29.2 yields a quadratic equation for $K$ whose solution gives the number of processors for the maximum speedup. For example, with $n = 2{,}000$ antiprotons, the maximum speedup comes at $K_{\text{best}} \approx 20$. Unlike the original program in Chapter 27, when computation and communication are overlapped, the speedup remains linear right up until $K = K_{\text{best}}$, and then the speedup decreases again. With overlapping, the maximum speedup for $n = 2{,}000$ is 20; without overlapping, the maximum speedup was only 9.891.

This computation time model assumes that all messages are inter-processor messages and would have to be modified if some messages are intra-processor messages.

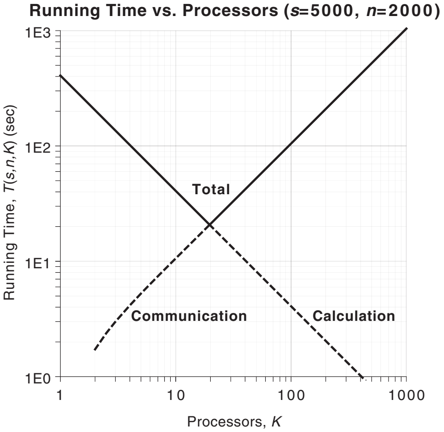


**Figure 29.6** $T(s,n,K)$ predicted by computation time model

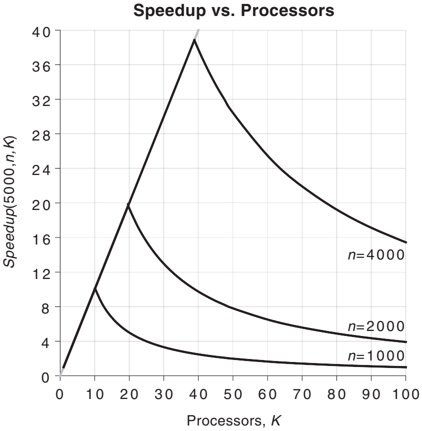


**Figure 29.7** $Speedup(5000,n,K)$ predicted by computation time model

## 29.5 Pipelined Overlapped Program Performance

Table 29.1 (at the end of the chapter) lists the AntiprotonClu3 program's performance on the "tardis" parallel computer. The program was run with the following command,

```
$ java -Dpj.np=$K edu.rit.clu.antimatter.AntiprotonClu3 \
  142857 7 0.00001 1000 5 $N outfile.dat
```

where the number of antiprotons (`$N`) was 1,000, 1,400, 2,000, 2,800, 4,000, or 5,600, the number of time steps between snapshots was 1,000, and the number of snapshots was 5, for a total of 5,000 time steps. For comparison, Figure 29.8 plots the pipelined nonoverlapped AntiprotonClu2 program's performance, and Figure 29.9 plots the pipelined overlapped AntiprotonClu3 program's performance. Overlapping the computation with the communication has definitely improved the program's performance.

We've taken the antiproton motion program's design about as far as we can. Further improvements in the program's performance would have to come from switching to a different algorithm, such as a higher-order integration algorithm.

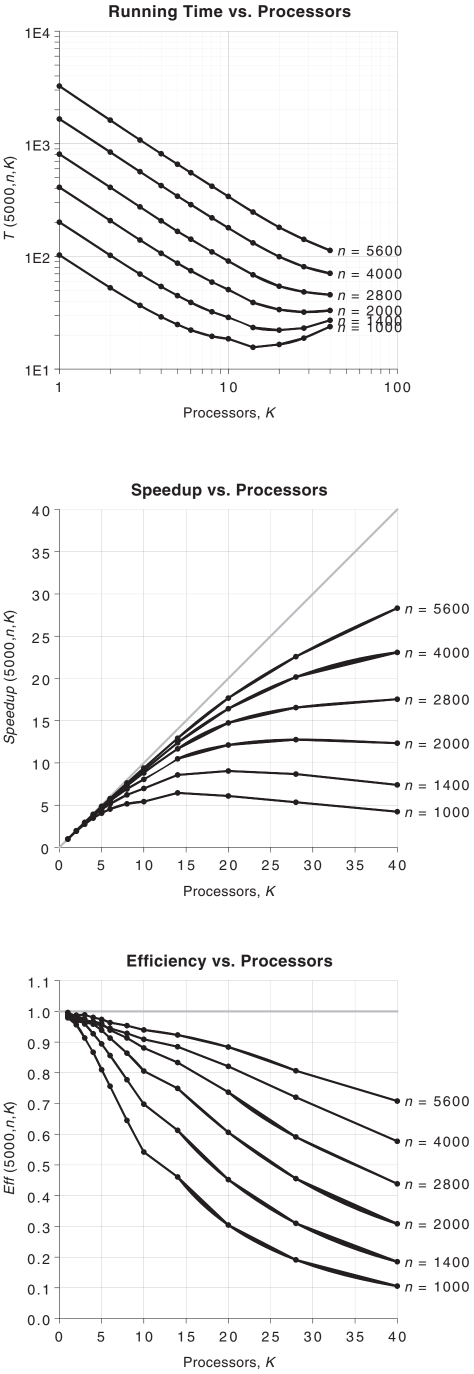


**Figure 29.8** AntiprotonSeq/AntiprotonClu running-time metrics, without overlapping

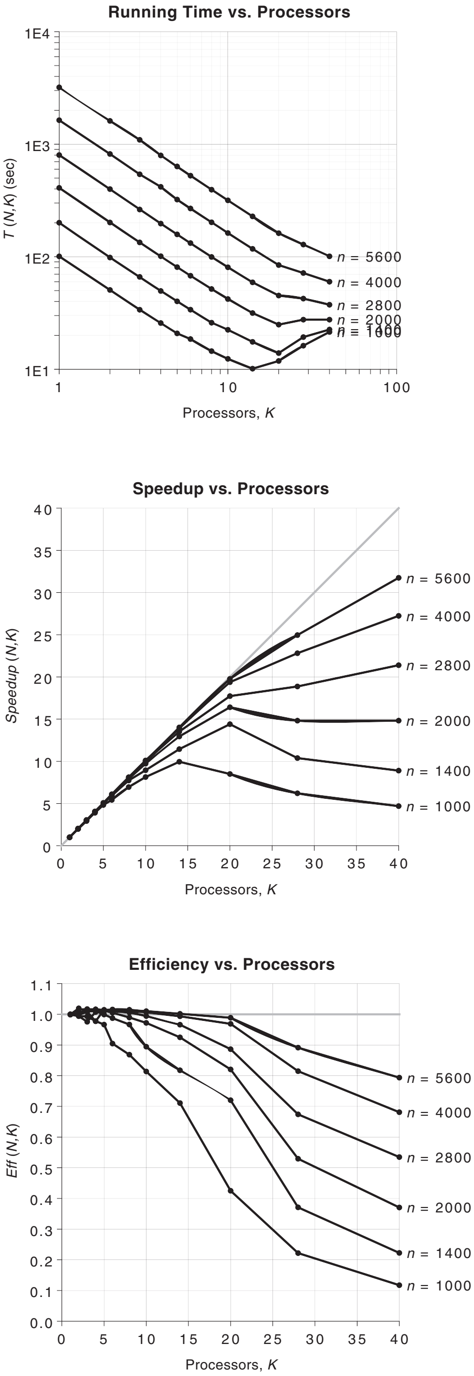


**Figure 29.9** AntiprotonSeq/AntiprotonClu3 running-time metrics, with overlapping

**Table 28.1** AntiprotonSeq/AntiprotonClu3 running-time metrics with $s$ = 5000 and $n$ varying

| *n* | *K* | *T* | *Spdup* | *Eff* | *EDSF* | *n* | *K* | *T* | *Spdup* | *Eff* | *EDSF* |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1000 | seq | 100699 | | | | 2800 | seq | 801136 | | | |
| 1000 | 1 | 100772 | 0.999 | 0.999 | | 2800 | 1 | 801086 | 1.000 | 1.000 | |
| 1000 | 2 | 50646 | 1.988 | 0.994 | 0.005 | 2800 | 2 | 399121 | 2.007 | 1.004 | -0.004 |
| 1000 | 3 | 33806 | 2.979 | 0.993 | 0.003 | 2800 | 3 | 262751 | 3.049 | 1.016 | -0.008 |
| 1000 | 4 | 25743 | 3.912 | 0.978 | 0.007 | 2800 | 4 | 197063 | 4.065 | 1.016 | -0.005 |
| 1000 | 5 | 20838 | 4.832 | 0.966 | 0.008 | 2800 | 5 | 158012 | 5.070 | 1.014 | -0.003 |
| 1000 | 6 | 18553 | 5.428 | 0.905 | 0.021 | 2800 | 6 | 132258 | 6.057 | 1.010 | -0.002 |
| 1000 | 8 | 14494 | 6.948 | 0.868 | 0.022 | 2800 | 8 | 99543 | 8.048 | 1.006 | -0.001 |
| 1000 | 10 | 12375 | 8.137 | 0.814 | 0.025 | 2800 | 10 | 80574 | 9.943 | 0.994 | 0.001 |
| 1000 | 14 | 10120 | 9.950 | 0.711 | 0.031 | 2800 | 14 | 59230 | 13.526 | 0.966 | 0.003 |
| 1000 | 20 | 11847 | 8.500 | 0.425 | 0.071 | 2800 | 20 | 45190 | 17.728 | 0.886 | 0.007 |
| 1000 | 28 | 16205 | 6.214 | 0.222 | 0.130 | 2800 | 28 | 42453 | 18.871 | 0.674 | 0.018 |
| 1000 | 40 | 21452 | 4.694 | 0.117 | 0.193 | 2800 | 40 | 37451 | 21.392 | 0.535 | 0.022 |
| 1400 | seq | 200554 | | | | 4000 | seq | 1634482 | | | |
| 1400 | 1 | 200671 | 0.999 | 0.999 | | 4000 | 1 | 1634533 | 1.000 | 1.000 | |
| 1400 | 2 | 98362 | 2.039 | 1.019 | -0.020 | 4000 | 2 | 818556 | 1.997 | 0.998 | 0.002 |
| 1400 | 3 | 66209 | 3.029 | 1.010 | -0.005 | 4000 | 3 | 540408 | 3.025 | 1.008 | -0.004 |
| 1400 | 4 | 49612 | 4.042 | 1.011 | -0.004 | 4000 | 4 | 418105 | 3.909 | 0.977 | 0.008 |
| 1400 | 5 | 40143 | 4.996 | 0.999 | 0.000 | 4000 | 5 | 322183 | 5.073 | 1.015 | -0.004 |
| 1400 | 6 | 33851 | 5.925 | 0.987 | 0.002 | 4000 | 6 | 268354 | 6.091 | 1.015 | -0.003 |
| 1400 | 8 | 25930 | 7.734 | 0.967 | 0.005 | 4000 | 8 | 202298 | 8.080 | 1.010 | -0.001 |
| 1400 | 10 | 22423 | 8.944 | 0.894 | 0.013 | 4000 | 10 | 162374 | 10.066 | 1.007 | -0.001 |
| 1400 | 14 | 17522 | 11.446 | 0.818 | 0.017 | 4000 | 14 | 117502 | 13.910 | 0.994 | 0.000 |
| 1400 | 20 | 13926 | 14.401 | 0.720 | 0.020 | 4000 | 20 | 84345 | 19.379 | 0.969 | 0.002 |
| 1400 | 28 | 19311 | 10.385 | 0.371 | 0.063 | 4000 | 28 | 71653 | 22.811 | 0.815 | 0.008 |
| 1400 | 40 | 22535 | 8.900 | 0.222 | 0.090 | 4000 | 40 | 60005 | 27.239 | 0.681 | 0.012 |
| 2000 | seq | 408964 | | | | 5600 | seq | 3202070 | | | |
| 2000 | 1 | 409467 | 0.999 | 0.999 | | 5600 | 1 | 3201937 | 1.000 | 1.000 | |
| 2000 | 2 | 201393 | 2.031 | 1.015 | -0.016 | 5600 | 2 | 1610633 | 1.988 | 0.994 | 0.006 |
| 2000 | 3 | 134042 | 3.051 | 1.017 | -0.009 | 5600 | 3 | 1094020 | 2.927 | 0.976 | 0.013 |
| 2000 | 4 | 101121 | 4.044 | 1.011 | -0.004 | 5600 | 4 | 794416 | 4.031 | 1.008 | -0.003 |
| 2000 | 5 | 80853 | 5.058 | 1.012 | -0.003 | 5600 | 5 | 634043 | 5.050 | 1.010 | -0.002 |
| 2000 | 6 | 67803 | 6.032 | 1.005 | -0.001 | 5600 | 6 | 525467 | 6.094 | 1.016 | -0.003 |
| 2000 | 8 | 51657 | 7.917 | 0.990 | 0.001 | 5600 | 8 | 394446 | 8.118 | 1.015 | -0.002 |
| 2000 | 10 | 42072 | 9.721 | 0.972 | 0.003 | 5600 | 10 | 316965 | 10.102 | 1.010 | -0.001 |
| 2000 | 14 | 31583 | 12.949 | 0.925 | 0.006 | 5600 | 14 | 228354 | 14.022 | 1.002 | 0.000 |
| 2000 | 20 | 24923 | 16.409 | 0.820 | 0.011 | 5600 | 20 | 161926 | 19.775 | 0.989 | 0.001 |
| 2000 | 28 | 27596 | 14.820 | 0.529 | 0.033 | 5600 | 28 | 128298 | 24.958 | 0.891 | 0.005 |
| 2000 | 40 | 27592 | 14.822 | 0.371 | 0.043 | 5600 | 40 | 100859 | 31.748 | 0.794 | 0.007 |

*This page intentionally left blank*

# CHAPTER 30 All-Reduce

in which we calculate the temperature throughout a metal plate; we encounter a program that needs the all-reduce message passing operation; and we learn one way to solve partial differential equations in parallel

# CHAPTER 30 All-Reduce

## 30.1 A Heat Distribution Problem

A thin metal plate has a temperature of 0 °C along each edge. At certain points in the interior of the plate, the temperature is a given value larger than 0 °C; these points are called "hot spots" (not to be confused with the HotSpot JIT compiler!). Question: What is the temperature everywhere else in the plate?

Figure 30.1 shows an example. At three interior points, and in a rectangular region near the bottom, the plate's temperature is 100 °C, while the edges are 0 °C. Points at 100 °C are white in the image; points at 0 °C are black; points at intermediate temperatures are shades of gray.

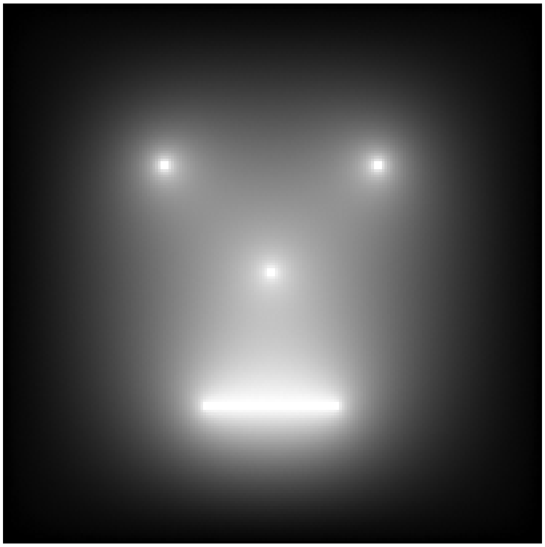


**Figure 30.1** A metal plate with hot spots

Let *h*(*x*,*y*) be the temperature at point (*x*,*y*). The function *h*(*x*,*y*) satisfies the following partial differential equation, known as **Laplace's equation**, at all points except the boundary and the hot spots:

$$\frac{\partial^2 h}{\partial x^2} + \frac{\partial^2 h}{\partial y^2} = 0 \qquad (30.1)$$

We will write a program that solves the partial differential equation, that is, a program that finds *h*(*x*,*y*) that satisfies Equation 30.1.

First, we'll divide the plate into a **mesh** of equally spaced points with *H* points in the *y* direction and *W* points in the *x* direction (Figure 30.2). In other words, *h* is an *H*×*W*-element matrix. We will solve for *h* only at these mesh points. By changing *H* and *W,* we can make the mesh resolution as coarse or as fine as we want. Instead of *h*(*x*,*y*), we'll use the notation *h*[*r*,*c*], where *r* and *c* are the row and column indexes of a certain mesh point, $1 \leq r \leq H$, $1 \leq c \leq W$(*r* corresponds to the *y* direction, *c* to the *x* direction).

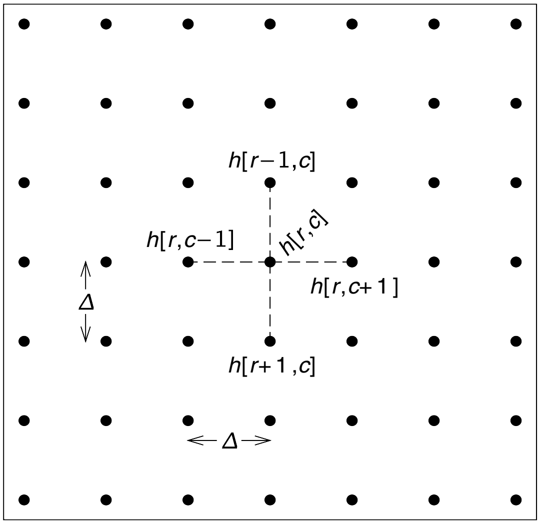


**Figure 30.2** Mesh points

Next, we need formulas for the second partial derivatives of *h* with respect to *x* and *y* at the mesh points. Consider the *x* direction. Let Δ be the distance between adjacent mesh points. Then the first partial derivative of *h* with respect to *x* between *h*[*r*,*c*–1] and *h*[*r*,*c*] is the slope of *h* between those two points:

$$\frac{\partial h_1}{\partial x} = \frac{h[r,c] - h[r,c-1]}{\Delta} \tag{30.2}$$

Likewise, the first partial derivative of *h* with respect to *x* between *h*[*r*,*c*] and *h*[*r*,*c*+1] is the slope of *h* between those two points:

$$\frac{\partial h_2}{\partial x} = \frac{h[r,c+1] - h[r,c]}{\Delta} \tag{30.3}$$

The second partial derivative of *h* with respect to *x* at *h*[*r*,*c*] is the slope of those two first partial derivatives:

$$\frac{\partial^2 h}{\partial x^2} = \frac{\partial h_2 / \partial x - \partial h_1 / \partial x}{\Delta} = \frac{h[r,c-1] + h[r,c+1] - 2h[r,c]}{\Delta^2} \tag{30.4}$$

A similar calculation gives the second partial derivative of *h* with respect to *y* at *h*[*r*,*c*]:

$$\frac{\partial h_1}{\partial y} = \frac{h[r,c] - h[r-1,c]}{\Delta} \tag{30.5}$$

$$\frac{\partial h_2}{\partial y} = \frac{h[r,+1,c] - h[r,c]}{\Delta} \tag{30.6}$$

$$\frac{\partial^2 h}{\partial y^2} = \frac{\partial h_2 / \partial y - \partial h_1 / \partial y}{\Delta} = \frac{h[r-1,c] + h[r+1,c] - 2h[r,c]}{\Delta^2} \tag{30.7}$$

Plugging (30.4) and (30.7) into (30.1) and multiplying both sides by $\Delta^2$ gives the following:

$$h[r-1,c] + h[r+1,c] + h[r,c-1] + h[r,c+1] - 4h[r,c] = 0 \tag{30.8}$$

And a little bit of algebra gives the following:

$$h[r,c] = \frac{h[r-1,c] + h[r+1,c] + h[r,c-1] + h[r,c+1]}{4} \tag{30.9}$$

Thus, the solution of the partial differential equation has every mesh point (except the boundaries and hot spots) at a temperature equal to the average of the four neighboring mesh points' temperatures.

**Initial program**. Our program will be based on (30.9). We begin by setting up the temperature matrix *h* with *H*+2 rows and *W*+2 columns, and every element initially 0. The elements in row 0, row *H*+1, column 0, and column *W*+1 are the boundary elements, which stay at 0. The hot spot elements are set to their given values. We update all the interior points in rows 1 through *H* and columns 1 through *W* using (30.9), leaving the boundary and hot spot points unchanged, and storing the new values in a separate matrix so as not to wipe out the old values before we're done using them. Then, we replace the old values with the new values, and repeat until a suitable termination criterion (to be specified later) is achieved. This is called a **relaxation** algorithm—as it iterates, the interior points gradually "relax" from the initial state and converge on the solution.

```
For all points $(r,c)$:
    $h[r,c] = 0$
For all hot spot points $(r,c)$:
    $h[r,c]$ = hot spot temperature
Do:
    For all points $(r,c) = (1,1)$ to $(H,W)$:
        If $(r,c)$ is not a hot spot point:
            $hnew[r,c] = (h[r-1,c] + h[r+1,c] + h[r,c-1] + h[r,c+1])/4$
    $h = hnew$
Until done
```

**Termination using residuals**. Let's do some more algebra on the update formula (30.9):

$$\begin{aligned} hnew[r,c] &= \frac{h[r-1,c]+h[r+1,c]+h[r,c-1]+h[r,c+1]}{4} \\ &= h[r,c] + \frac{h[r-1,c]+h[r+1,c]+h[r,c-1]+h[r,c+1]}{4} - h[r,c] \\ &= h[r,c] + \frac{h[r-1,c]+h[r+1,c]+h[r,c-1]+h[r,c+1]-4h[r,c]}{4} \end{aligned} \tag{30.10}$$

Define a mesh point's **residual** $\xi[r,c]$ as the term in the numerator:

$$\xi[r,c] = h[r-1,c]+h[r+1,c]+h[r,c-1]+h[r,c+1]-4h[r,c] \tag{30.11}$$

Then the update formula becomes the following:

$$hnew[r,c] = h[r,c] + \frac{\xi[r,c]}{4} \tag{30.12}$$

Thus, the residual (when scaled by 1/4) represents the difference between the old and the new values of the mesh point.

We define the update formula as (30.12) instead of (30.9) because the residual gives us an especially convenient termination criterion. Let the **total absolute residual** $|\xi|$ be the sum of the absolute values of the residuals for all the interior points (except hot spots):

$$|\xi| = \sum_{r=1}^{H} \sum_{c=1}^{W} |\xi[r,c]| \tag{30.13}$$

Then $|\xi|$ tells how far the current solution is from the ultimate solution. At the beginning, $|\xi|$ is a large value. As we do more iterations, $|\xi|$ decreases. If we continued iterating forever, $|\xi|$ would go to zero. Instead of iterating forever, we will stop when $|\xi|$ reaches some fraction $\varepsilon$ of its initial value, for example, $\varepsilon = 0.001$.

One further practical consideration is that we should put an upper limit on the number of iterations, to make sure the program stops even if $|\xi|$ is not converging. If the program ever hits this limit, it aborts and reports it did not converge to a solution. For the moment, we will leave the upper limit unspecified. Putting it all together:

```
$MAX$ = t.b.d.
$EPS$ = 0.001
For all points $(r,c)$:
    $h[r,c] = 0$
For all hot spot points $(r,c)$:
    $h[r,c]$ = hot spot temperature
$initialTotalAbsXi = 0$
For all points $(r,c) = (1,1)$ to $(H,W)$:
    If $(r,c)$ is not a hot spot point:
        $xi = h[r–1,c] + h[r+1,c] + h[r,c–1] + h[r,c+1] – 4*h[r,c]$
        $initialTotalAbsXi$ += abs($xi$)
$iterations = 0$
Do:
    $totalAbsXi = 0$
    For all points $(r,c) = (1,1)$ to $(H,W)$:
        If $(r,c)$ is not a hot spot point:
            $xi = h[r–1,c] + h[r+1,c] + h[r,c–1] + h[r,c+1] – 4*h[r,c]$
            $totalAbsXi$ += abs($xi$)
            $hnew[r,c]$ += $xi/4$
    $h = hnew$
    ++ $iterations$
Until $iterations == MAX$ or $totalAbsXi < EPS*initialTotalAbsXi$
If $iterations == MAX$:
    Abort (“Did not converge”)
```

**Red-Black Mesh Updating**. Imagine that the mesh points are colored alternating red and black, like a checkerboard (Figure 30.3). In terms of indexes, $h[r,c]$ is a red point if $r+c$ is odd, and $h[r,c]$ is a black point if $r+c$ is even.

Then, to compute the new value of a red point, we need only look at the current values of the four surrounding black points, and vice versa to compute the new value of a black point. This suggests that we can do away with the separate mesh holding the new values. Instead, we can do one half-sweep over the mesh, updating the red points in place using the current values of the black points. Then we can do another half-sweep over the mesh, updating the black points in place using the just-computed values of the red points.

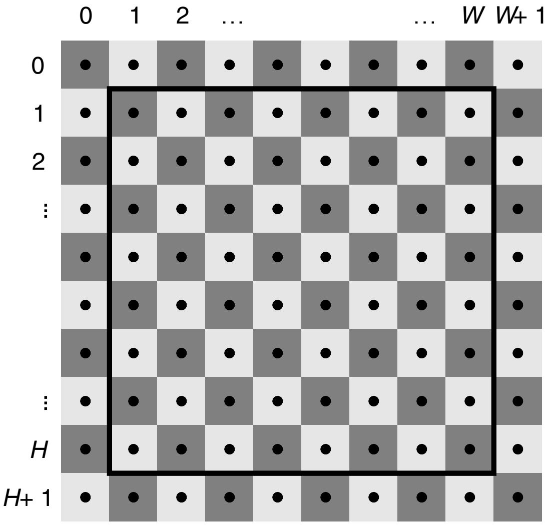


**Figure 30.3** Red-black mesh points

During each half-sweep, we still iterate over all the rows from $r = 1$ to $H$, but we only iterate over half the columns. For the red half-sweep, the column iteration starts at $c = 1$ for an even-numbered row or $c = 2$ for an odd-numbered row (see Figure 30.3). An expression for the initial index is $c = 1 + (r\&1)$, where & is the bitwise Boolean and operator. $(r\&1)$ is 0 if $r$ is even and 1 if $r$ is odd, and adding 1 to that gives 1 if $r$ is even and 2 if $r$ is odd. For the black half-sweep, the column iteration starts at $c = 2$ for an even-numbered row or $c = 1$ for an odd-numbered row. An expression for the initial index is $c = 2 – (r\&1)$. In both cases, the column stride is 2.

```
MAX = t.b.d.
EPS = 0.001
For all points (r,c):
    h[r,c] = 0
For all hot spot points (r,c):
    h[r,c] = hot spot temperature
initialTotalAbsXi = 0
For all points (r,c) = (1,1) to (H,W):
    If (r,c) is not a hot spot point:
        xi = h[r–1,c] + h[r+1,c] + h[r,c–1] + h[r,c+1] – 4*h[r,c]
        initialTotalAbsXi += abs(xi)
iterations = 0
Do:
    totalAbsXi = 0
    // Red half-sweep
```

```
        For $r$ = 1 to $H$:
            For $c$ = 1 + ($r$&1) to $W$ stride 2:
                If ($r$,$c$) is not a hot spot point:
                    $xi = h[r-1,c] + h[r+1,c] + h[r,c-1] + h[r,c+1] - 4*h[r,c]$
                    $totalAbsXi$ += abs($xi$)
                    $h[r,c]$ += $xi/4$
        // Black half-sweep
        For $r$ = 1 to H:
            For $c$ = 2 – ($r$&1) to $W$ stride 2:
                If ($r$,$c$) is not a hot spot point:
                    $xi = h[r-1,c] + h[r+1,c] + h[r,c-1] + h[r,c+1] - 4*h[r,c]$
                    $totalAbsXi$ += abs($xi$)
                    $h[r,c]$ += $xi/4$
        ++ $iterations$
    Until $iterations$ == $MAX$ or $totalAbsXi < EPS*initialTotalAbsXi$
    If $iterations$ == $MAX$:
        Abort (“Did not converge”)
```

**Successive overrelaxation**. As stated so far, the relaxation algorithm takes a long time to converge on the solution. In fact, with an $n$×$n$ mesh, the number of iterations required is $O(n^2)$, and with each iteration requiring $O(n^2)$ operations, the whole algorithm's running time is $O(n^4)$. The problem is that with update rule (30.12), only a small correction $\xi[r,c]/4$ is added to each mesh point $h[r,c]$ at each iteration, so it takes many iterations to achieve convergence.

To address this problem, suppose we *overcorrect* each mesh point at each iteration,

$$hnew[r,c] = h[r,c] + \omega\frac{\xi[r,c]}{4} \tag{30.14}$$

using a **relaxation parameter** $\omega$ greater than 1. In effect, we are anticipating several iterations' worth of corrections and adding them in all at once. This should reduce the total number of iterations required to reach convergence.

The next question is what value of $\omega > 1$ to use. If ω is too small, it won't reduce the number of iterations very much. If $\omega$ is too large, each mesh point will overshoot the solution, requiring more iterations to bring it back to convergence. Somewhere in there is an optimum value for $\omega$—not too small, not too large.

The optimum relaxation parameter is related to the **spectral radius** $\rho_s$. The spectral radius is a characteristic of the relaxation algorithm that tells how quickly the mesh points relax to the final solution. For an $H$×$W$ mesh, the spectral radius is the following:

$$\rho_s = \frac{\cos(\pi / H) + \cos(\pi / W)}{2} \tag{30.15}$$

And the optimum relaxation parameter is the following:

$$\omega = \frac{2}{1+\sqrt{1-\rho_s^2}} \tag{30.16}$$

With this value of $\omega$, the relaxation algorithm—now called a **successive overrelaxation (SOR)** algorithm—requires only $O(n)$ iterations to reach convergence instead of $O(n^2)$, for a total running time of $O(n^3)$ instead of $O(n^4)$. Now we can set the maximum number of iterations to be $O(n)$; a value of $2(H+W)$ works well.

```
MAX = 2*(H+W)
EPS = 0.001
rho_s = (cos(π/H) + cos(π/W))/2
omega = 2 / (1 + sqrt(1 – rho_s2))
For all points (r,c):
    h[r,c] = 0
For all hot spot points (r,c):
    h[r,c] = hot spot temperature
initialTotalAbsXi = 0
For all points (r,c) = (1,1) to (H,W):
    If (r,c) is not a hot spot point:
        xi = h[r–1,c] + h[r+1,c] + h[r,c–1] + h[r,c+1] – 4*h[r,c]
        initialTotalAbsXi += abs(xi)
iterations = 0
Do:
    totalAbsXi = 0
    // Red half-sweep
    For r = 1 to H:
        For c = 1 + (r&1) to W stride 2:
            If (r,c) is not a hot spot point:
                xi = h[r–1,c] + h[r+1,c] + h[r,c–1] + h[r,c+1] – 4*h[r,c]
                totalAbsXi += abs(xi)
                h[r,c] += omega*xi/4
    // Black half-sweep
    For r = 1 to H:
        For c = 2 – (r&1) to W stride 2:
            If (r,c) is not a hot spot point:
                xi = h[r–1,c] + h[r+1,c] + h[r,c–1] + h[r,c+1] – 4*h[r,c]
                totalAbsXi += abs(xi)
                h[r,c] += omega*xi/4
    ++ iterations
Until iterations == MAX or totalAbsXi < EPS*initialTotalAbsXi
If iterations == MAX:
    Abort (“Did not converge”)
```

**Chebyshev acceleration**. One last improvement reduces the number of iterations still further. It turns out that fewer iterations are needed if the relaxation parameter is allowed to change as the algorithm progresses. $\omega$ should start out at 1 and gradually increase according to the following rules. For the first iteration, red half-sweep:

$$\omega \leftarrow 1 \tag{30.17}$$

For the first iteration, black half-sweep:

$$\omega \leftarrow \frac{1}{1-\rho_s^2/2} \tag{30.18}$$

Thereafter, the new value of ω is computed from the old value of ω using this formula:

$$\omega \leftarrow \frac{1}{1-\rho_s^2\omega/4} \tag{30.19}$$

As the number of iterations increases, ω approaches its optimum value (30.16). Adding this to our pseudocode, we get our final algorithm—**successive overrelaxation with Chebyshev acceleration**.

```
MAX = 2*(H+W)
EPS = 0.001
rho_s = (cos(π/H) + cos(π/W))/2
omega = 1
For all points (r,c):
    h[r,c] = 0
For all hot spot points (r,c):
    h[r,c] = hot spot temperature
initialTotalAbsXi = 0
For all points (r,c) = (1,1) to (H,W):
    If (r,c) is not a hot spot point:
        xi = h[r–1,c] + h[r+1,c] + h[r,c–1] + h[r,c+1] – 4*h[r,c]
        initialTotalAbsXi += abs(xi)
iterations = 0
Do:
    totalAbsXi = 0
    // Red half-sweep
    For r = 1 to H:
        For c = 1 + (r&1) to W stride 2:
            If (r,c) is not a hot spot point:
                xi = h[r–1,c] + h[r+1,c] + h[r,c–1] + h[r,c+1] – 4*h[r,c]
                totalAbsXi += abs(xi)
                h[r,c] += omega*xi/4
    If iterations == 0:
        $\mathit{omega} = 1 / (1 - \mathit{rho\_s}^2/2)$
    Else:
        $\mathit{omega} = 1 / (1 - \mathit{rho\_s}^2 * \mathit{omega}/4)$
    // Black half-sweep
    For $r = 1$ to $H$:
        For $c = 2 - (r \& 1)$ to $W$ stride 2:
            If $(r,c)$ is not a hot spot point:
                $\mathit{xi} = h[r-1,c] + h[r+1,c] + h[r,c-1] + h[r,c+1] - 4*h[r,c]$
                $\mathit{totalAbsXi}$ += abs($\mathit{xi}$)
                $h[r,c]$ += $\mathit{omega}*\mathit{xi}/4$
    $\mathit{omega} = 1 / (1 - \mathit{rho\_s}^2 * \mathit{omega}/4)$
    ++ $\mathit{iterations}$
Until $\mathit{iterations} == \mathit{MAX}$ or $\mathit{totalAbsXi} < \mathit{EPS} * \mathit{initialTotalAbsXi}$
If $\mathit{iterations} == \mathit{MAX}$:
    Abort ("Did not converge")
```

## 30.2 Sequential Heat Distribution Program

Class edu.rit.clu.heat.HotSpotSeq is a sequential program that solves the heat distribution problem. The command-line arguments are the following:

- *imagefile,* the output PJG image file name.
- *H*, the number of mesh rows (not including the boundaries).
- *W*, the number of mesh columns (not including the boundaries).
- *rl*, the lower row index of a hot spot.
- *cl*, the lower column index of a hot spot.
- *ru*, the upper row index of a hot spot.
- *cu*, the upper column index of a hot spot.
- *temp*, the temperature of a hot spot.

Every mesh element in the rectangular region from [*rl*, *cl*] through [*ru*, *cu*] is part of a hot spot with temperature *temp*. Multiple hot spots may be specified by giving *rl*, *cl*, *ru*, *cu*, and *temp* for each hot spot.

Class HotSpotSeq produces a color image where each pixel's hue is proportional to the plate's temperature, with 0 °C being blue and 100 °C being red. Figure 30.1 was produced by class HotSpotGray, which produces a grayscale image but is otherwise identical to class HotSpotSeq. The command that produced Figure 30.1 (compare to Figure 30.4) was the following:

```
$ java edu.rit.clu.heat.HotSpotGray fig_a.pjg 200 200 \
  59 59 61 61 100 59 139 61 141 100 99 99 101 101 100 \
  149 75 151 125 100
```

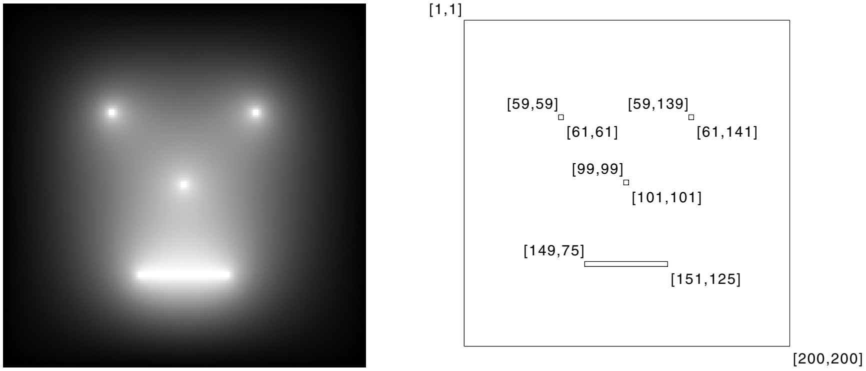


**Figure 30.4** Hot spot coordinates in Figure 30.1

Here is the source code for class edu.rit.clu.heat.HotSpotSeq.

```
package edu.rit.clu.heat;
import edu.rit.color.HSB;
import edu.rit.image.ColorImageRow;
import edu.rit.image.PJGHueImage;
import edu.rit.image.PJGImage;
import edu.rit.pj.Comm;
import java.io.BufferedOutputStream;
import java.io.File;
import java.io.FileOutputStream;
public class HotSpotSeq
   {
   private static final double MIN_TEMP = 0.0;
   private static final double MAX_TEMP = 100.0;
   private static final double DELTA_TEMP = MAX_TEMP - MIN_TEMP;

   private static final double MIN_HUE = 4.0/6.0;
   private static final double MAX_HUE = 0.0;
   private static final double DELTA_HUE = MAX_HUE - MIN_HUE;

   private static final double EPS = 1.0e-3;

   // Command line arguments.
   static File imagefile;
   static int H;
   static int W;
```

`h[r][c]` holds the temperature of mesh point *h*[*r*,*c*]. `hotspot[r][c]` is true if *h*[*r*,*c*] is part of a hot spot, and is false otherwise.

```
   // Temperature mesh.
   static double[][] h;

   // Mesh of hot spot locations.
   static boolean[][] hotspot;

   // Variables for total absolute residual.
   static double EPS_initialTotalAbsXi;
   static double totalAbsXi;

   // Other variables used in the successive overrrelaxation
   // algorithm.
   static int MAX;
   static double rho_s_sqr;
   static double omega_over_4;
   static int iterations;

   /**
    * Main program.
    */
   public static void main
      (String[] args)
      throws Exception
      {
      // Start timing.
      long t1 = System.currentTimeMillis();

      // Parse command line arguments.
      if (args.length < 8 || (args.length % 5) != 3) usage();
      imagefile = new File (args[0]);
      H = Integer.parseInt (args[1]);
      W = Integer.parseInt (args[2]);
      if (H < 1) usage();
      if (W < 1) usage();

      // Initialize temperature and hot spot meshes.
      h = new double [H+2] [W+2];
      hotspot = new boolean [H+2] [W+2];
```

Each element of h is initialized to 0 by default. Each element of hotspot is initialized to false by default. Here, we set each hot spot element of h to the proper temperature and set each hot spot element of hotspot to true.

```
// Record hot spot coordinates and temperatures.
int n = (args.length - 3) / 5;
for (int i = 0; i < n; ++ i)
   {
   int rl = Integer.parseInt (args[3+5*i]);
   int cl = Integer.parseInt (args[4+5*i]);
   int ru = Integer.parseInt (args[5+5*i]);
   int cu = Integer.parseInt (args[6+5*i]);
   double temp = Double.parseDouble (args[7+5*i]);
   if (1 > rl || rl > W) usage();
   if (1 > cl || cl > H) usage();
   if (1 > ru || ru > W) usage();
   if (1 > cu || cu > H) usage();
   if (MIN_TEMP > temp || temp > MAX_TEMP) usage();
   for (int r = rl; r <= ru; ++ r)
      {
      double[] h_r = h[r];
      boolean[] hotspot_r = hotspot[r];
      for (int c = cl; c <= cu; ++ c)
         {
         h_r[c] = temp;
         hotspot_r[c] = true;
         }
      }
   }
```

By including extra boundary rows and columns in h and hotspot, we can loop r from 1 to *H* and simply refer to rows r-1 and r+1 without needing special cases for the boundaries; and similarly for c.

```
// Compute initial total absolute residual, then multiply by
// EPS.
totalAbsXi = 0.0;
double xi;
for (int r = 1; r <= H; ++ r)
   {
   double[] h_rm1 = h[r-1];
   double[] h_r   = h[r];
   double[] h_rp1 = h[r+1];
   boolean[] hotspot_r = hotspot[r];
   for (int c = 1; c <= W; ++ c)
      {
```

```
      xi = hotspot_r[c] ? 0.0 :
         h_rm1[c]+h_rp1[c]+h_r[c-1]+h_r[c+1]-4.0*h_r[c];
      totalAbsXi += Math.abs (xi);
      }
   }
EPS_initialTotalAbsXi = EPS * totalAbsXi;

// Initialize other variables.
MAX = 2 * (W + H);
rho_s_sqr = 0.5 * (Math.cos(Math.PI/W)+Math.cos(Math.PI/H));
rho_s_sqr = rho_s_sqr * rho_s_sqr;
omega_over_4 = 0.25;
iterations = 0;

long t2 = System.currentTimeMillis();

// Perform successive overrelaxation.
do
   {
   totalAbsXi = 0.0;

   // Red half-sweep.
   for (int r = 1; r <= H; ++ r)
      {
      double[] h_rm1 = h[r-1];
      double[] h_r   = h[r];
      double[] h_rp1 = h[r+1];
      boolean[] hotspot_r = hotspot[r];
      for (int c = 1 + (r&1); c <= W; c += 2)
         {
         xi = hotspot_r[c] ? 0.0 :
            h_rm1[c]+h_rp1[c]+h_r[c-1]+h_r[c+1]-4.0*h_r[c];
         totalAbsXi += Math.abs (xi);
         h_r[c] += omega_over_4 * xi;
         }
      }
   omega_over_4 = 0.25 /
      (1.0-rho_s_sqr*(iterations==0 ? 0.5 : omega_over_4));

   // Black half-sweep.
   for (int r = 1; r <= H; ++ r)
      {
      double[] h_rm1 = h[r-1];
      double[] h_r   = h[r];
      double[] h_rp1 = h[r+1];
```

```
            boolean[] hotspot_r = hotspot[r];
            for (int c = 2 - (r&1); c <= W; c += 2)
               {
               xi = hotspot_r[c] ? 0.0 :
                  h_rm1[c]+h_rp1[c]+h_r[c-1]+h_r[c+1]-4.0*h_r[c];
               totalAbsXi += Math.abs (xi);
               h_r[c] += omega_over_4 * xi;
               }
            }
         omega_over_4 = 0.25 / (1.0-rho_s_sqr*omega_over_4);

         ++ iterations;
         }
      while (iterations < MAX &&
         totalAbsXi >= EPS_initialTotalAbsXi);

      // Check for convergence.
      if (iterations == MAX)
         {
         System.err.println ("HotSpotSeq: Did not converge");
         System.exit (1);
         }

      long t3 = System.currentTimeMillis();
```

We calculate each pixel's hue and store the results in an integer matrix. Then, we write the pixel data to a PJG image file using class edu.rit.image.PJGHueImage. Class PJGHueImage uses a different compression algorithm from class PJGColorImage. The former is better suited for color images with a continuous range of hues; the latter is better suited for color images with a small number of discrete colors.

```
      // Generate image.
      int[][] matrix = new int [H+2] [W+2];
      ColorImageRow matrix_r = new ColorImageRow (matrix[0]);
      for (int r = 0; r <= H+1; ++ r)
         {
         double[] h_r = h[r];
         matrix_r.setArray (matrix[r]);
         for (int c = 0; c <= W+1; ++ c)
            {
            matrix_r.setPixelHSB
               (/*c  */ c,
                /*hue*/ (float)
                  ((h_r[c]-MIN_TEMP)/DELTA_TEMP*DELTA_HUE+MIN_HUE),
                /*sat*/ 1.0f,
                /*bri*/ 1.0f);
            }
         }
      PJGHueImage image = new PJGHueImage (H+2, W+2, matrix);
      PJGImage.Writer writer =
         image.prepareToWrite
            (new BufferedOutputStream
               (new FileOutputStream (imagefile)));
      writer.write();
      writer.close();

      // Stop timing.
      long t4 = System.currentTimeMillis();
      System.out.println (iterations + " iterations");
      System.out.println ((t2-t1) + " msec pre");
      System.out.println ((t3-t2) + " msec calc");
      System.out.println ((t4-t3) + " msec post");
      System.out.println ((t4-t1) + " msec total");
      }
   }
```

## 30.3 Collective Communication: All-Reduce

Our first task in designing a cluster parallel version of the heat distribution program is to partition the computation among the $K$ parallel processes. As we have done with all the other cluster parallel programs that calculate a matrix, we will slice the `h` and `hotspot` matrices into $K$ slices by rows. Each process will calculate the mesh points in a subrange of the rows and all the columns. Because the calculations take the same time in every loop iteration, dividing the matrices into equal-sized slices will yield a balanced load.

Consider the calculation of the initial total absolute residual $|\xi|$. Each process calculates $|\xi|$ for *its own slice* of the matrix. But each process needs to know $|\xi|$ for the *entire* matrix to know when to terminate the iterations. The partial values of $|\xi|$ from each process must be combined; this is a *reduction* with addition as the reduction operator. But the result of the reduction must end up in all the processes, not in just one root process. This is the **all-reduce** collective communication operation.

```
world.allReduce (buf, op);
```

The `allReduce()` method's first argument is a buffer referring to the value or values from each process that are to be reduced together. The second argument is the reduction operator to use, such as `DoubleOp.SUM`. After the `allReduce()` method returns, the contents of the buffer in every process have been replaced with the results of the reduction.

The cluster parallel heat distribution program needs to do an all-reduce operation in two places: to determine the initial value of $|\xi|$; and to determine the value of $|\xi|$ after each pair of half-sweeps to decide whether to stop iterating.

An all-reduce is equivalent to a reduction followed by a broadcast. If implemented that way, an all-reduce would require ($2 \log_2 K$) message rounds, where $K$ is the number of processes. However, an all-reduce can be done in only ($\log_2 K$) message rounds (Figure 30.5). In each round, each process exchanges its data buffer with another process, and then it combines its own data with the incoming data using the reduction operator. In the first round, processes one rank apart exchange data; in the second round, processes two ranks apart exchange data; in the third round, processes four ranks apart exchange data; and so on. This message pattern is called a "butterfly" because of the crisscrossing pattern the messages make.

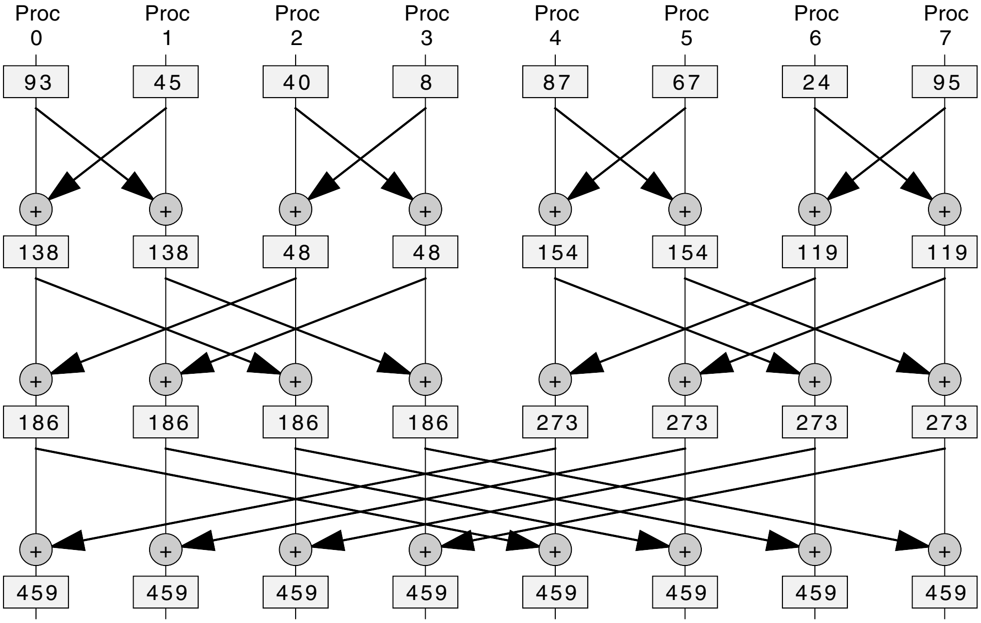


**Figure 30.5** All-reduce among eight processes, sum as the reduction operator

## 30.4 Mesh Element Allocation and Communication

As already mentioned, each process in the cluster parallel program "owns" one slice of the `h` matrix. But, to update the mesh elements in its own slice, the process must refer to the mesh elements in the last row of the previous process's slice and to the mesh elements in the first row of the next process's slice. This suggests that each process should allocate the rows in its own slice of the `h` matrix, plus *two additional rows*, one at the top and one at the bottom. These additional rows will hold copies of the mesh elements from the last row of the previous process's slice and from the first row of the next process's slice.

Figure 30.6 shows an example with a mesh of $H = 8$ rows and $W = 8$ columns, sliced among $K = 4$ processes. The h matrix actually comprises 10 rows and 10 columns, with boundary elements of value 0 in row 0, row 9, column 0, and column 9. Process 0 allocates its slice consisting of rows 1–2, with the extra rows 0 and 3. Process 1 allocates its slice consisting of rows 3–4, with the extra rows 2 and 5. Process 2 allocates its slice consisting of rows 5–6, with the extra rows 4 and 7. Process 3 allocates its slice consisting of rows 7–8, with the extra rows 6 and 9. (In Figures 30.6–30.9, the mesh elements' values are for illustration only and do not represent the actual values the program would compute.)

After each process has performed the red half-sweep, the newly computed red element values must be communicated to the adjacent processes in preparation for the upcoming black half-sweep. This is done with two rounds of communication. The first round sends data forward (Figure 30.6). Each process sends the red elements in the last row of its slice to the next process; simultaneously, each process receives the red elements from the last row of the previous process's slice, and stores them in the extra row before the process's own slice. These two transfers are done with one send-receive operation.

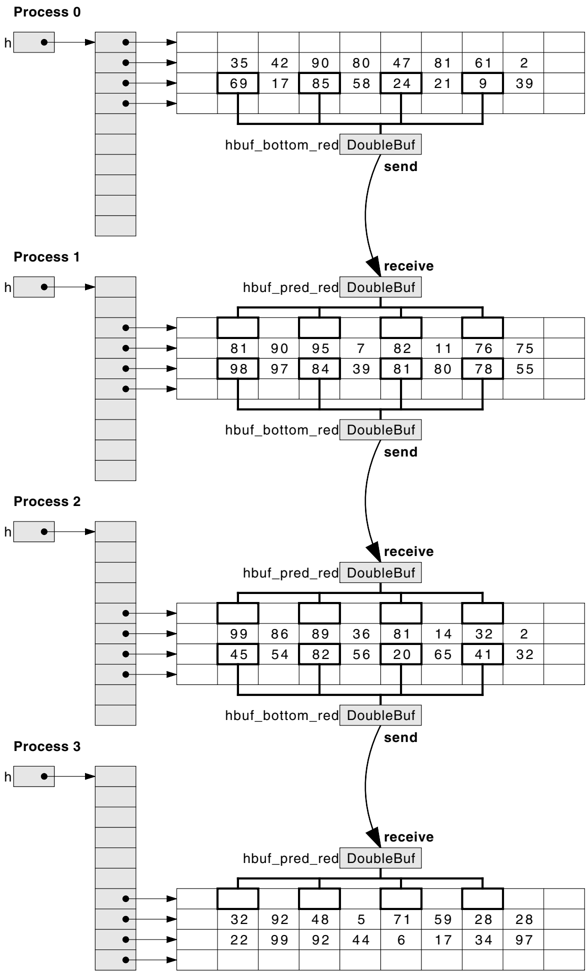


**Figure 30.6** First round of communication after red half-sweep

(However, process 0 only sends, and process *K*–1 only receives.) The second round sends data backward (Figure 30.7). Each process sends the red elements in the first row of its slice to the previous process; simultaneously, each process receives the red elements from the first row of the next process's slice, and stores them in the extra row after the process's own slice. Again, these two transfers are done with one send-receive operation. (However, process 0 only receives, and process *K*–1 only sends.)

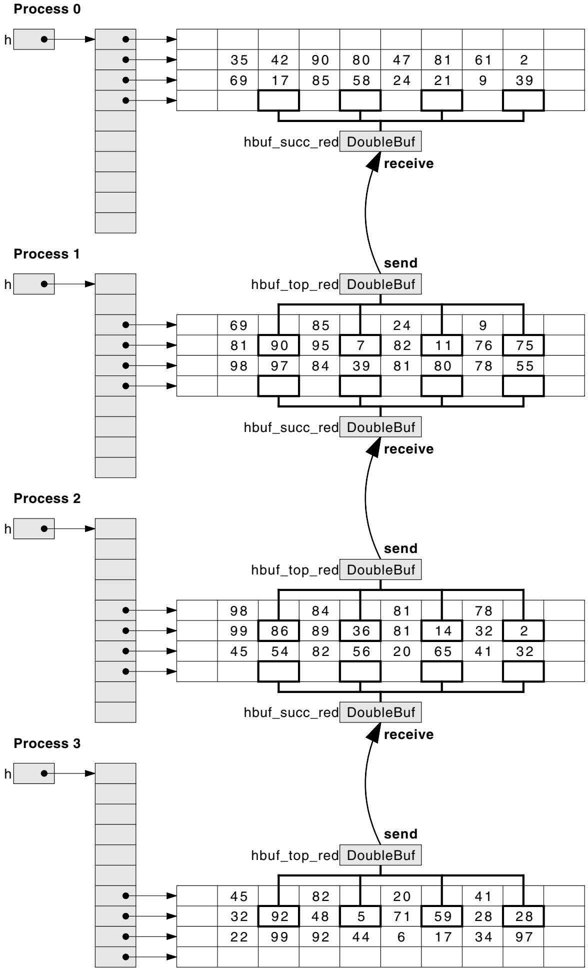


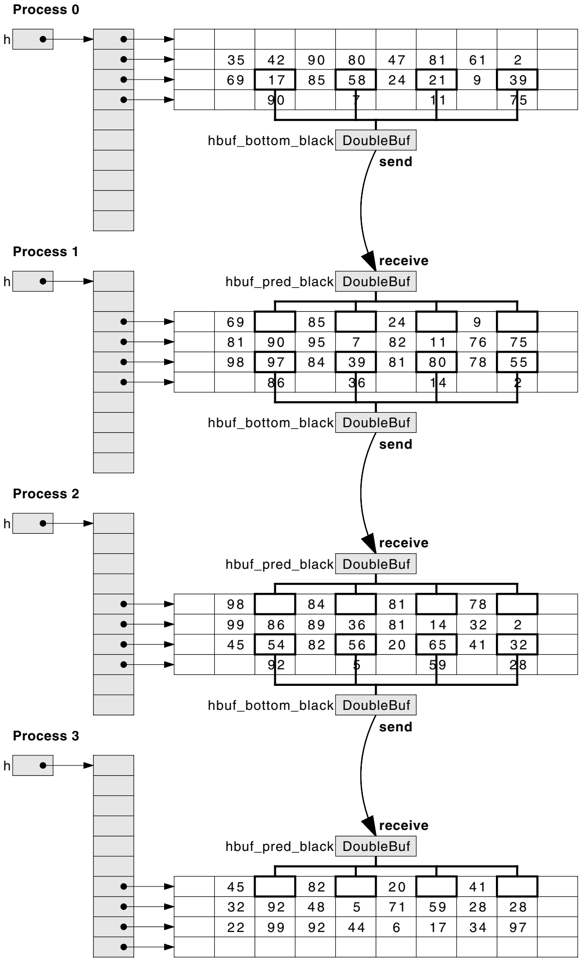
**Figure 30.7** Second round of communication after red half-sweep

Note that the buffers for these communication operations refer to *noncontiguous* elements in one row of the `h` matrix, namely the red elements. To send the entire row would unnecessarily increase the communication time. To create a buffer referring to just the red elements in row `r` of the `h` matrix, specify a range with a stride of 2.

```
    DoubleBuf.sliceBuffer (h[r], new Range (1+(r&1), W, 2));
```

Once the red elements have been transferred, the black half-sweep can commence. After each process has performed the black half-sweep, the newly computed black element values must be communicated to the adjacent processes in preparation for the upcoming red half-sweep. Again, this is done with two rounds of communication. The first round sends data forward (Figure 30.8); the second round sends data backward (Figure 30.9). The only difference is that this time, the buffers refer to the black elements.

```
    DoubleBuf.sliceBuffer (h[r], new Range (2-(r&1), W, 2));
```



Process 0

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | |
| | 35 | 42 | 90 | 80 | 47 | 81 | 61 | 2 | |
| | 69 | 17 | 85 | 58 | 24 | 21 | 9 | 39 | |
| | | 90 | | 7 | | 11 | | 75 | |

Process 1

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | 69 | | 85 | | 24 | | 9 | | |
| | 81 | 90 | 95 | 7 | 82 | 11 | 76 | 75 | |
| | 98 | 97 | 84 | 39 | 81 | 80 | 78 | 55 | |
| | | 86 | | 36 | | 14 | | 2 | |

Process 2

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | 98 | | 84 | | 81 | | 78 | | |
| | 99 | 86 | 89 | 36 | 81 | 14 | 32 | 2 | |
| | 45 | 54 | 82 | 56 | 20 | 65 | 41 | 32 | |
| | | 92 | | 5 | | 59 | | 28 | |

Process 3

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | 45 | | 82 | | 20 | | 41 | | |
| | 32 | 92 | 48 | 5 | 71 | 59 | 28 | 28 | |
| | 22 | 99 | 92 | 44 | 6 | 17 | 34 | 97 | |
| | | | | | | | | | |

**Figure 30.8** First round of communication after black half-sweep

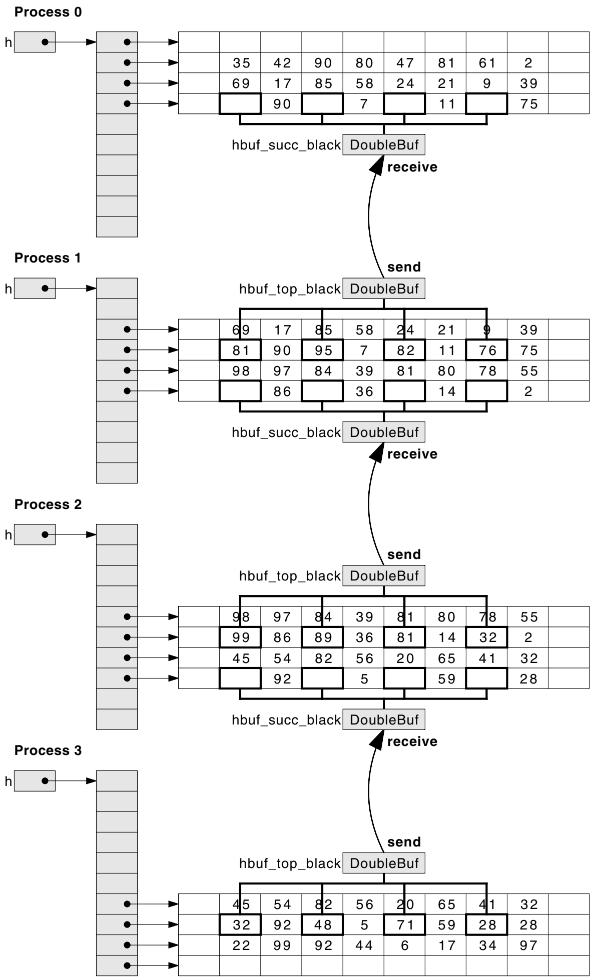


**Figure 30.9** Second round of communication after black half-sweep

## 30.5 Parallel Heat Distribution Program

Here is the source code for class edu.rit.clu.heat.HotSpotClu, the cluster parallel version of the heat distribution program. The command-line arguments are the same as the sequential version.

```
package edu.rit.clu.heat;
import edu.rit.color.HSB;
import edu.rit.image.ColorImageRow;
import edu.rit.image.PJGHueImage;
import edu.rit.image.PJGImage;
import edu.rit.io.Files;
import edu.rit.mp.DoubleBuf;
import edu.rit.mp.buf.DoubleItemBuf;
import edu.rit.pj.Comm;
import edu.rit.pj.reduction.DoubleOp;
import edu.rit.util.Arrays;
import edu.rit.util.Range;
import java.io.BufferedOutputStream;
import java.io.File;
import java.io.FileOutputStream;
public class HotSpotClu
   {
   private static final double MIN_TEMP = 0.0;
   private static final double MAX_TEMP = 100.0;
   private static final double DELTA_TEMP = MAX_TEMP - MIN_TEMP;

   private static final double MIN_HUE = 4.0/6.0;
   private static final double MAX_HUE = 0.0;
   private static final double DELTA_HUE = MAX_HUE - MIN_HUE;

   private static final double EPS = 1.0e-3;
```

The following constants are used to decide which message passing operations the process will do.

```
   private static final int FIRST  = 0;
   private static final int MIDDLE = 1;
   private static final int LAST   = 2;
   private static final int SINGLE = 3;

   // World communicator.
   static Comm world;
   static int size;
   static int rank;
   static int position;
   static int predRank;
   static int succRank;

   // Command line arguments.
   static File imagefile;
   static int H;
   static int W;
```

The following variables store the bounds of this process's slice of the `h` matrix.

```
   // Row slice index ranges.
   static Range[] slices;
   static Range mySlice;
   static int myLb;
   static int myUb;
   static int myLen;
   // Temperature grid.
   static double[][] h;

   // Mesh of hot spot locations.
   static boolean[][] hotspot;

   // Variables for total absolute residual.
   static double EPS_initialTotalAbsXi;
   static double totalAbsXi;

   // Other variables used in the successive overrrelaxation
   // algorithm.
   static int MAX;
   static double rho_s_sqr;
   static double omega_over_4;
   static int iterations;
```

Here are the buffers used for message passing. The first one is used to all-gather the total absolute residual. The others are used to transfer elements of the h matrix (see Figures 30.6–30.9).

```
   // Communication buffers.
   static DoubleItemBuf xibuf;
   static DoubleBuf hbuf_pred_red;
   static DoubleBuf hbuf_pred_black;
   static DoubleBuf hbuf_top_red;
   static DoubleBuf hbuf_top_black;
   static DoubleBuf hbuf_bottom_red;
   static DoubleBuf hbuf_bottom_black;
   static DoubleBuf hbuf_succ_red;
   static DoubleBuf hbuf_succ_black;

   /**
    * Main program.
    */
   public static void main
      (String[] args)
      throws Exception
      {
      // Start timing.
      long t1 = System.currentTimeMillis();

      // Initialize world communicator.
      Comm.init (args);
      world = Comm.world();
      size = world.size();
      rank = world.rank();
```

We determine if this process is the only process (SINGLE), if it is the first process (FIRST), if it is the last process (LAST), or if it is neither the first nor the last (MIDDLE). We also determine the rank of the previous process and the rank of the next process.

```
if (size == 1) position = SINGLE;
else if (rank == 0) position = FIRST;
else if (rank < size-1) position = MIDDLE;
else position = LAST;
predRank = rank - 1;
succRank = rank + 1;

// Parse command line arguments.
if (args.length < 8 || (args.length % 5) != 3) usage();
imagefile = new File (args[0]);
H = Integer.parseInt (args[1]);
W = Integer.parseInt (args[2]);
if (H < 1) usage();
if (W < 1) usage();
```

We determine the lower and upper bounds of this process's slice of the h matrix.

```
// Determine row slice index ranges.
slices = new Range (1, H) .subranges (size);
mySlice = slices[rank];
myLb = mySlice.lb();
myUb = mySlice.ub();
myLen = mySlice.length();
```

We allocate storage just for the rows within those bounds, plus the extra rows above and below, in the h and hotspot matrices.

```
// Initialize temperature and hot spot meshes.
h = new double [H+2] [];
Arrays.allocate (h, new Range (myLb-1, myUb+1), W+2);
hotspot = new boolean [H+2] [];
Arrays.allocate (hotspot, new Range (myLb-1, myUb+1), W+2);

// Record hot spot coordinates and temperatures.
int n = (args.length - 3) / 5;
for (int i = 0; i < n; ++ i)
   {
```

```
            int rl = Integer.parseInt (args[3+5*i]);
            int cl = Integer.parseInt (args[4+5*i]);
            int ru = Integer.parseInt (args[5+5*i]);
            int cu = Integer.parseInt (args[6+5*i]);
            double temp = Double.parseDouble (args[7+5*i]);
            if (1 > rl || rl > W) usage();
            if (1 > cl || cl > H) usage();
            if (1 > ru || ru > W) usage();
            if (1 > cu || cu > H) usage();
            if (MIN_TEMP > temp || temp > MAX_TEMP) usage();
            for (int r = rl; r <= ru; ++ r)
               {
               double[] h_r = h[r];
               boolean[] hotspot_r = hotspot[r];
               if (h_r != null)
                  {
                  for (int c = cl; c <= cu; ++ c)
                     {
                     h_r[c] = temp;
                     hotspot_r[c] = true;
                     }
                  }
               }
            }
```

The `redBuffer()` and `blackBuffer()` subroutines create buffers for noncontiguous elements of the given row of the `h` matrix.

```
         // Initialize communication buffers.
         xibuf = DoubleBuf.buffer();
         hbuf_pred_red = redBuffer (myLb-1);
         hbuf_pred_black = blackBuffer (myLb-1);
         hbuf_top_red = redBuffer (myLb);
         hbuf_top_black = blackBuffer (myLb);
         hbuf_bottom_red = redBuffer (myUb);
         hbuf_bottom_black = blackBuffer (myUb);
         hbuf_succ_red = redBuffer (myUb+1);
         hbuf_succ_black = blackBuffer (myUb+1);

         // Compute initial total absolute residual, then multiply by
         // EPS.
         totalAbsXi = 0.0;
         double xi;
```

To compute the initial partial $|\xi|$ value, we loop over only the rows in this process's slice.

```
      for (int r = myLb; r <= myUb; ++ r)
          {
         double[] h_rm1 = h[r-1];
         double[] h_r   = h[r];
         double[] h_rp1 = h[r+1];
         boolean[] hotspot_r = hotspot[r];
         for (int c = 1; c <= W; ++ c)
            {
            xi =
               hotspot_r[c] ?
                  0.0 :
                  h_rm1[c]+h_rp1[c]+h_r[c-1]+h_r[c+1]-4.0*h_r[c];
            totalAbsXi += Math.abs (xi);
            }
         }
```

Here is the all-reduce operation that adds up the initial partial $|\xi|$ values from all the processes and returns the initial total $|\xi|$ to all the processes.

```
      xibuf.item = totalAbsXi;
      world.allReduce (xibuf, DoubleOp.SUM);
      totalAbsXi = xibuf.item;
      EPS_initialTotalAbsXi = EPS * totalAbsXi;

      // Initialize other variables.
      MAX = 2 * (W + H);
      rho_s_sqr = 0.5 * (Math.cos(Math.PI/W)+Math.cos(Math.PI/H));
      rho_s_sqr = rho_s_sqr * rho_s_sqr;
      omega_over_4 = 0.25;
      iterations = 0;

      long t2 = System.currentTimeMillis();

      // Perform successive overrelaxation.
      do
         {
         totalAbsXi = 0.0;
```

To do the red half-sweep, we loop over only the rows in this process's slice.

```
            // Red half-sweep.
            for (int r = myLb; r <= myUb; ++ r)
               {
               double[] h_rm1 = h[r-1];
               double[] h_r   = h[r];
               double[] h_rp1 = h[r+1];
               boolean[] hotspot_r = hotspot[r];
               for (int c = 1 + (r&1); c <= W; c += 2)
                  {
                  xi = hotspot_r[c] ? 0.0 :
                     h_rm1[c]+h_rp1[c]+h_r[c-1]+h_r[c+1]-4.0*h_r[c];
                  totalAbsXi += Math.abs (xi);
                  h_r[c] += omega_over_4 * xi;
                  }
               }
            omega_over_4 = 0.25 /
               (1.0-rho_s_sqr*(iterations==0 ? 0.5 : omega_over_4));
```

Here are the two rounds of communication after the red half-sweep (Figures 30.6 and 30.7). If there is only one process (case `SINGLE`), then no communication occurs. Otherwise, the first process (case `FIRST`) sends, and then receives; the last process (case `LAST`) receives, and then sends; the other processes (case `MIDDLE`) do two send-receives.

```
            // Exchange boundary row red cells with neighboring
            // processes.
            switch (position)
               {
               case FIRST:
                  world.send (succRank, hbuf_bottom_red);
                  world.receive (succRank, hbuf_succ_red);
                  break;
               case MIDDLE:
                  world.sendReceive
                     (succRank, hbuf_bottom_red,
                      predRank, hbuf_pred_red);
                  world.sendReceive
                     (predRank, hbuf_top_red,
                      succRank, hbuf_succ_red);
                  break;
               case LAST:
                  world.receive (predRank, hbuf_pred_red);
                  world.send (predRank, hbuf_top_red);
                  break;
               }
```

The black half-sweep mirrors the red half-sweep.

```
// Black half-sweep.
for (int r = myLb; r <= myUb; ++ r)
   {
   double[] h_rm1 = h[r-1];
   double[] h_r   = h[r];
   double[] h_rp1 = h[r+1];
   boolean[] hotspot_r = hotspot[r];
   for (int c = 2 - (r&1); c <= W; c += 2)
      {
      xi = hotspot_r[c] ? 0.0 :
         h_rm1[c]+h_rp1[c]+h_r[c-1]+h_r[c+1]-4.0*h_r[c];
      totalAbsXi += Math.abs (xi);
      h_r[c] += omega_over_4 * xi;
      }
   }
omega_over_4 = 0.25 / (1.0 - rho_s_sqr * omega_over_4);

// Exchange boundary row black cells with neighboring
// processes.
switch (position)
   {
   case FIRST:
      world.send (succRank, hbuf_bottom_black);
      world.receive (succRank, hbuf_succ_black);
      break;
   case MIDDLE:
      world.sendReceive
         (succRank, hbuf_bottom_black,
          predRank, hbuf_pred_black);
      world.sendReceive
         (predRank, hbuf_top_black,
          succRank, hbuf_succ_black);
      break;
   case LAST:
      world.receive (predRank, hbuf_pred_black);
      world.send (predRank, hbuf_top_black);
      break;
   }
```

Here is the other all-reduce operation that adds up the partial $|\xi|$ values from all the processes and returns the total $|\xi|$ to all the processes.

```
         // Determine total absolute residual from all processes.
         xibuf.item = totalAbsXi;
         world.allReduce (xibuf, DoubleOp.SUM);
         totalAbsXi = xibuf.item;

         ++ iterations;
         }
      while (iterations < MAX &&
         totalAbsXi >= EPS_initialTotalAbsXi);

      // Check for convergence.
      if (iterations == MAX)
         {
         System.err.println ("HotSpotClu: Did not converge");
         System.exit (1);
         }

      long t3 = System.currentTimeMillis();
```

Class HotSpotClu uses the *parallel output files* pattern to store just the process's own slice of the image in the output PJG file. The separate image file slices can later be combined into a single image, if desired.

```
      // Generate image.
      int[][] matrix = new int [H+2] [];
      int rlb = rank == 0 ? myLb-1 : myLb;
      int rub = rank == size-1 ? myUb+1 : myUb;
      Arrays.allocate (matrix, new Range (rlb, rub), W+2);
      ColorImageRow matrix_r = new ColorImageRow (matrix[rlb]);
      for (int r = rlb; r <= rub; ++ r)
         {
         double[] h_r = h[r];
         matrix_r.setArray (matrix[r]);
         for (int c = 0; c <= W+1; ++ c)
            {
            matrix_r.setPixelHSB
               (/*c  */ c,
                /*hue*/ (float)
                  ((h_r[c]-MIN_TEMP)/DELTA_TEMP*DELTA_HUE+MIN_HUE),
                /*sat*/ 1.0f,
                /*bri*/ 1.0f);
            }
         }
      PJGHueImage image = new PJGHueImage (H+2, W+2, matrix);
      PJGImage.Writer writer =
         image.prepareToWrite
            (new BufferedOutputStream
               (new FileOutputStream
                  (Files.fileForRank (imagefile, rank))));
      writer.writeRowSlice (new Range (rlb, rub));
      writer.close();

      // Stop timing.
      long t4 = System.currentTimeMillis();
      System.out.println (iterations + " iterations " + rank);
      System.out.println ((t2-t1) + " msec pre " + rank);
      System.out.println ((t3-t2) + " msec calc " + rank);
      System.out.println ((t4-t3) + " msec post " + rank);
      System.out.println ((t4-t1) + " msec total " + rank);
      }

   /**
    * Returns a communication buffer for the red columns of the
    * given row of the h matrix.
    */
   private static DoubleBuf redBuffer (int r)
      {
      return DoubleBuf.sliceBuffer
         (h[r], new Range (1+(r&1), W, 2));
      }

   /**
    * Returns a communication buffer for the black columns of the
    * given row of the h matrix.
    */
   private static DoubleBuf blackBuffer (int r)
      {
      return DoubleBuf.sliceBuffer
         (h[r], new Range (2-(r&1), W, 2));
      }
   }
```

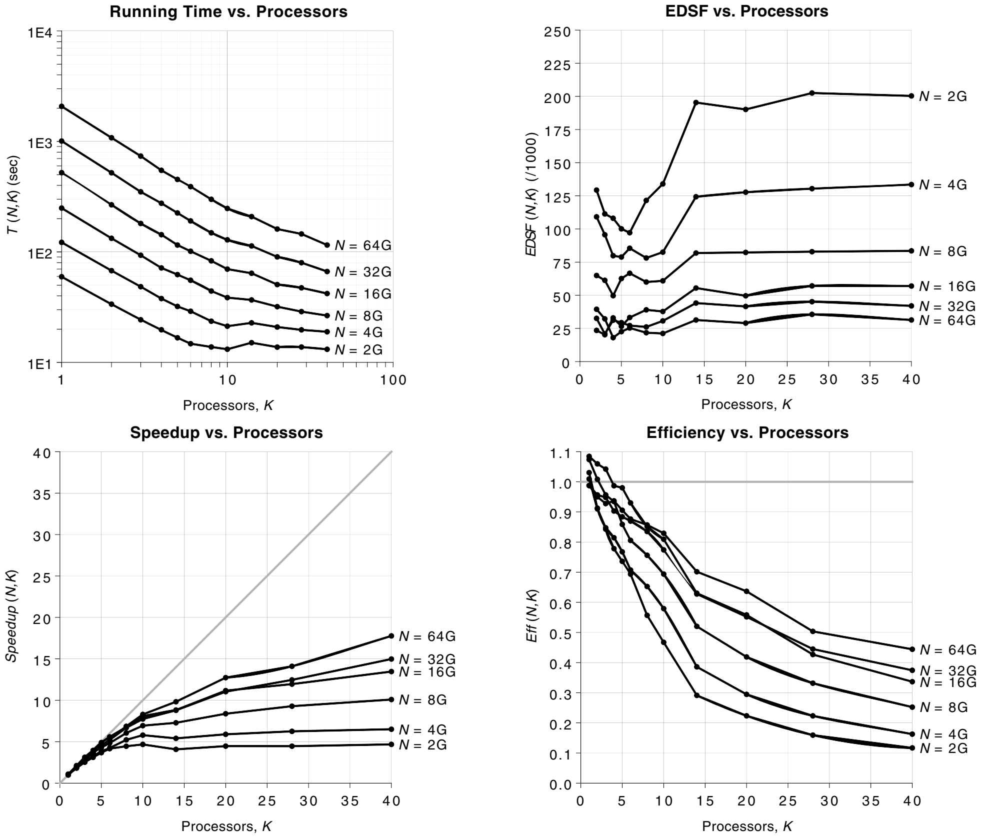


**Figure 30.10** HotSpotSeq/HotSpotClu running-time metrics

## 30.6 Parallel Program Performance

Table 30.1 (at the end of the chapter) lists, and Figure 30.10 plots, the HotSpotClu program's performance on the "tardis" parallel computer. The program was run with the following command:

```
$ java -Dpj.np=$K edu.rit.clu.heat.HotSpotClu out.pjg $n $n \
  $L $L $U $U 100
```

The program calculated an *n*×*n*-element mesh with one central hot spot at a temperature of 100 °C occupying coordinates (*L*,*L*) to (*U*,*U*). Because the successive overrelaxation algorithm's running time is $O(n^3)$, the problem size $N = n^3$. The following values of *n*, *N*, *L*, and *U* were used:

| *n* | *N* | *L* | *U* |
|---|---|---|---|
| 1260 | 2G | 614 | 646 |
| 1590 | 4G | 775 | 815 |
| 2000 | 8G | 975 | 1025 |
| 2520 | 16G | 1228 | 1292 |
| 3180 | 32G | 1550 | 1630 |
| 4000 | 64G | 1950 | 2050 |

Unlike the last few cluster parallel programs we've studied, the running-time metrics show that the HotSpotClu program does not experience a slowdown as the number of processes *K* increases. Rather, for a given problem size, the speedup approaches a limit as *K* increases, as predicted by Amdahl's Law for a constant sequential fraction.

It's fairly easy to see where the constant sequential fraction comes from. During each outer loop iteration of the successive overrelaxation algorithm, each process sends two rows' worth of data to other processes and simultaneously receives two rows' worth of data from other processes. While the time to send this data depends on the problem size (specifically, on the mesh width *W*), the time to send this data is the same no matter how many processes there are, because all the processes are sending at the same time. Although the time needed to do the all-reduce does increase as *K* increases, the all-reduce is communicating only one value. Because the row element transfers are communicating several thousand values, the additional time to do the all-reduce is nearly unnoticeable. Therefore, the sequential fraction appears constant.

The *EDSF* curves also show that the sequential fraction decreases as the problem size increases. This is due to the *surface-to-volume effect*. During each outer loop iteration, the computation time is $O(n^2)$, but the communication time is only $O(n)$, so the sequential fraction is $O(n^{-1})$. As the problem size scales up, the sequential fraction goes down; consequently, the speedups and efficiencies improve. However, to achieve good speedups and efficiencies all the way out to 40 processors, we would have to run the program with problem sizes several times larger than the ones measured here.

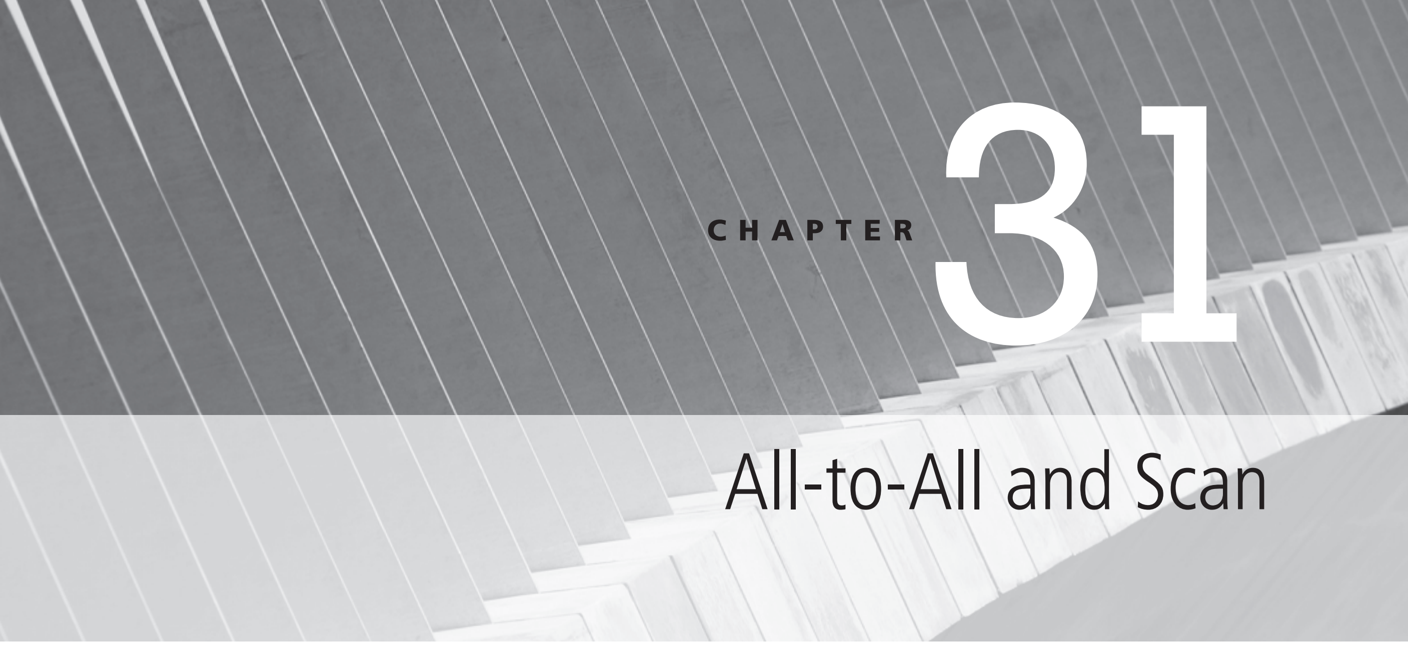
## 30.7 For Further Information

On numerical solution of partial differential equations using SOR:

- W. Press, S. Teukolsky, W. Vetterling, and B. Flannery. *Numerical Recipes: The Art of Scientific Computing, Third Edition*. Cambridge University Press, 2008, Chapter 20 and Section 20.5.

**Table 30.1** HotSpotSeq/HotSpotClu running-time metrics

| *N* | *K* | *T* | *Spdup* | *Eff* | *EDSF* | *N* | *K* | *T* | *Spdup* | *Eff* | *EDSF* |
|---:|---:|---:|---:|---:|---:|---:|---:|---:|---:|---:|---:|
| 2G | seq | 61469 | | | | 16G | seq | 565595 | | | |
| 2G | 1 | 59649 | 1.031 | 1.031 | | 16G | 1 | 521585 | 1.084 | 1.084 | |
| 2G | 2 | 33681 | 1.825 | 0.913 | 0.129 | 16G | 2 | 266921 | 2.119 | 2.119 | 0.023 |
| 2G | 3 | 24311 | 1.528 | 0.843 | 0.111 | 16G | 3 | 180901 | 3.127 | 3.127 | 0.020 |
| 2G | 4 | 19747 | 3.113 | 0.778 | 0.108 | 16G | 4 | 143278 | 3.948 | 3.948 | 0.033 |
| 2G | 5 | 16706 | 3.679 | 0.736 | 0.100 | 16G | 5 | 115414 | 4.901 | 4.091 | 0.027 |
| 2G | 6 | 14722 | 4.161 | 0.694 | 0.097 | 16G | 6 | 101370 | 5.580 | 5.580 | 0.033 |
| 2G | 8 | 13799 | 4.455 | 0.557 | 0.122 | 16G | 8 | 82999 | 6.814 | 6.814 | 0.039 |
| 2G | 10 | 13157 | 4.672 | 0.467 | 0.134 | 16G | 10 | 69883 | 8.093 | 8.093 | 0.038 |
| 2G | 14 | 15081 | 4.076 | 0.291 | 0.195 | 16G | 14 | 64108 | 8.823 | 8.823 | 0.055 |
| 2G | 20 | 13757 | 4.468 | 0.223 | 0.190 | 16G | 20 | 50661 | 11.164 | 11.164 | 0.050 |
| 2G | 28 | 13780 | 4.461 | 0.159 | 0.203 | 16G | 28 | 47328 | 11.951 | 11.951 | 0.057 |
| 2G | 40 | 13143 | 4.677 | 0.117 | 0.200 | 16G | 40 | 42005 | 13.465 | 13.465 | 0.057 |
| 4G | seq | 123205 | | | | 32G | seq | 995382 | | | |
| 4G | 1 | 122052 | 1.009 | 1.009 | | 32G | 1 | 1007863 | 0.988 | 0.988 | |
| 4G | 2 | 67687 | 1.820 | 0.910 | 0.109 | 32G | 2 | 520348 | 1.913 | 1.913 | 0.033 |
| 4G | 3 | 45465 | 2.542 | 0.847 | 0.096 | 32G | 3 | 349967 | 2.844 | 2.844 | 0.021 |
| 4G | 4 | 37821 | 3.258 | 0.814 | 0.080 | 32G | 4 | 275552 | 3.612 | 3.612 | 0.031 |
| 4G | 5 | 32108 | 3.837 | 0.767 | 0.079 | 32G | 5 | 225254 | 4.419 | 4.419 | 0.029 |
| 4G | 6 | 29035 | 4.243 | 0.707 | 0.085 | 32G | 6 | 190789 | 5.217 | 5.217 | 0.027 |
| 4G | 8 | 23599 | 5.221 | 0.653 | 0.078 | 32G | 8 | 148995 | 6.681 | 6.681 | 0.026 |
| 4G | 10 | 21271 | 5.792 | 0.579 | 0.083 | 32G | 10 | 128594 | 7.741 | 7.741 | 0.031 |
| 4G | 14 | 22805 | 5.403 | 0.386 | 0.124 | 32G | 14 | 113284 | 8.787 | 8.787 | 0.044 |
| 4G | 20 | 20910 | 5.892 | 0.295 | 0.128 | 32G | 20 | 90178 | 11.038 | 11.038 | 0.042 |
| 4G | 28 | 19708 | 6.252 | 0.223 | 0.130 | 32G | 28 | 79897 | 12.458 | 12.458 | 0.045 |
| 4G | 40 | 13143 | 4.677 | 0.117 | 0.134 | 32G | 40 | 66457 | 14.978 | 14.978 | 0.042 |
| 8G | seq | 267691 | | | | 64G | seq | 267691 | | | |
| 8G | 1 | 249316 | 1.074 | 1.074 | | 64G | 1 | 2077124 | 0.987 | 0.987 | |
| 8G | 2 | 132748 | 2.017 | 1.008 | 0.065 | 64G | 2 | 1079537 | 1.900 | 1.900 | 0.039 |
| 8G | 3 | 93288 | 2.870 | 0.957 | 0.061 | 64G | 3 | 735966 | 2.783 | 2.783 | 0.032 |
| 8G | 4 | 71619 | 3.738 | 0.934 | 0.050 | 64G | 4 | 547380 | 3.747 | 3.747 | 0.018 |
| 8G | 5 | 62352 | 4.293 | 0.859 | 0.063 | 64G | 5 | 452957 | 4.528 | 4.528 | 0.023 |
| 8G | 6 | 55389 | 4.833 | 0.805 | 0.067 | 64G | 6 | 390011 | 5.259 | 5.259 | 0.025 |
| 8G | 8 | 44251 | 6.049 | 0.756 | 0.060 | 64G | 8 | 299199 | 6.855 | 6.855 | 0.022 |
| 8G | 10 | 38590 | 6.937 | 0.694 | 0.061 | 64G | 10 | 247331 | 8.292 | 8.292 | 0.021 |
| 8G | 14 | 36757 | 7.283 | 0.520 | 0.082 | 64G | 14 | 208810 | 9.822 | 9.822 | 0.031 |
| 8G | 20 | 31959 | 8.376 | 0.419 | 0.082 | 64G | 20 | 161101 | 12.731 | 12.731 | 0.029 |
| 8G | 28 | 28827 | 9.286 | 0.332 | 0.083 | 64G | 28 | 145454 | 14.101 | 14.101 | 0.036 |
| 8G | 40 | 13143 | 4.677 | 0.117 | 0.083 | 64G | 40 | 155433 | 17.768 | 17.768 | 0.031 |



# CHAPTER 31

# All-to-All and Scan

in which we learn about the Kolmogorov-Smirnov statistical test of randomness; we see how a block cipher can become a pseudorandom number generator; we develop a program to apply the Kolmogorov-Smirnov test to the AES block cipher; and we encounter a parallel program that needs the all-to-all and scan collective communication operations

# CHAPTER 31 All-to-All and Scan

## 31.1 The Kolmogorov-Smirnov Test

In Chapter 14, we introduced the notion of a **statistical test** of a pseudorandom number generator (PRNG). A statistical test usually goes like this:

1. Generate a large sample of random numbers.
2. Calculate a “statistic” from the sampled numbers. Let *D* be the value of the statistic.
3. Determine the statistic’s “*p* value.” This is the probability that the statistic’s value would be greater than or equal to *D* if the sampled numbers came from a truly random source.
4. If *p* is too small or too large, the PRNG fails the test, otherwise the PRNG passes the test.

Various statistical tests differ in the statistics they calculate and in the probability distributions of their statistics. Figure 31.1 shows an example of a probability density function for a statistic; the more likely statistic values have higher probability densities. For a given statistic value *D*, *p* is the area under the probability density function to the right of *D*. As *D* increases, *p* starts at 1 and decreases to 0. If *p* is too small (Figure 31.1) or too large (Figure 31.2), it indicates that *D* is not a likely value for the statistic; the probability of *D* occurring is too low, if the samples are drawn from a truly random source. Therefore, we can conclude that the source is not truly random, and the PRNG fails the test. A typical failure criterion is $p < 0.001$ or $p > 0.999$ (the Crush test suite uses these limits).

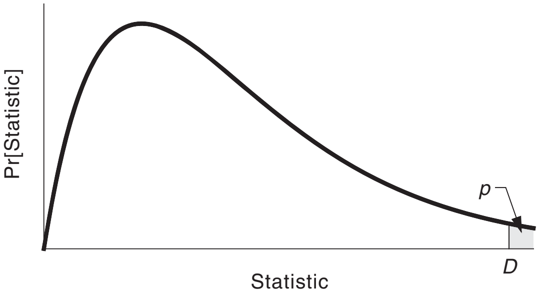


**Figure 31.1** Statistical test with *p* too small

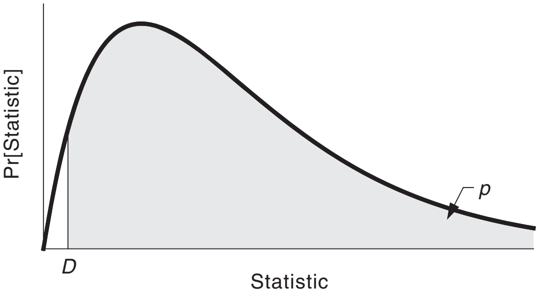


**Figure 31.2** Statistical test with *p* too large

In this chapter, we will focus on a particular statistical test, the **Kolmogorov-Smirnov (K-S) test**, defined by Russian mathematician Andrei Nikolaevich Kolmogorov in 1933 and refined by Nikolai Vasilievich Smirnov in 1939. Rather than describe the K-S test in all its generality, we will define it for the case of a random number source with a uniform distribution between 0 and 1, such as a PRNG algorithm.

Following the usual procedure for a statistical test, the first step is to generate some random numbers from the PRNG under test. For example, here are $n = 10$ random numbers from class edu.rit.util.Random. (Normally, we would generate millions or billions of random numbers.)

0.35612 0.42731 0.90112 0.80018 0.47976
0.81107 0.61478 0.02314 0.69704 0.17270

The second step is to compute the K-S statistic $D$. Figure 31.3 shows how $D$ is calculated. First we sort the random numbers into ascending order:

0.02314 0.17270 0.35612 0.42731 0.47976
0.61478 0.69704 0.80018 0.81107 0.90112

Next, we determine the cumulative distribution function for the random sample (the black curve in Figure 31.3). This starts at 0 and jumps up by $1/n$ at each sampled number.

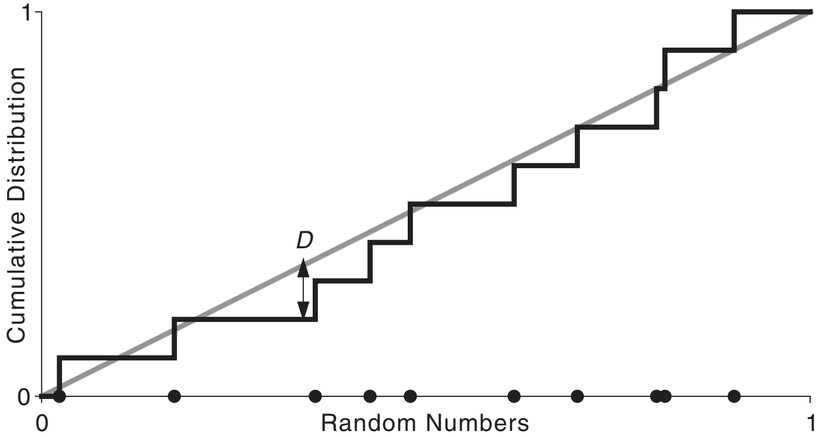


**Figure 31.3** Computing the K-S statistic *D* for a uniform PRNG

We compare the random sample's cumulative distribution function to the cumulative distribution function for a uniform random variable. The latter is just a straight line from (0, 0) to (1, 1) (the gray curve in Figure 31.3). The K-S statistic $D$ is the maximum absolute difference between the two cumulative distribution functions. Specifically, for each random number $x_i$, $0 \le i \le n{-}1$ (with the random numbers sorted), define the lower and upper differences as the following:

$$D_i^- = \left| \frac{i}{n} - x_i \right| \tag{31.1}$$

$$D_i^+ = \left| \frac{i+1}{n} - x_i \right| \tag{31.2}$$

Then $D$ is the largest of all the lower and upper differences. In our example, $D = 0.15612$ and it occurs as the lower difference for the third random number.

The third step in the K-S test is to calculate the $p$ value. This is given by the following formula:

$$p = P_{KS}\left( \left[ \sqrt{n} + 0.12 + \frac{0.11}{\sqrt{n}} \right] \cdot D \right) \tag{31.3}$$

where the function $P_{KS}$ is defined as the following:

$$P_{KS}(u) = 2\sum_{i=1}^{\infty} (-1)^{i-1} \exp(-2i^2u^2) \tag{31.4}$$

In our example, $p = 0.95136$.

The fourth and final step is to decide if the PRNG failed the test. According to the failure criterion ($p < 0.001$ or $p > 0.999$), class edu.rit.util.Random passes the K-S test for this tiny sample.

## 31.2 Block Ciphers as PRNGs

In Chapter 14, we saw one way to design a PRNG: feed a counter through a hash function. The hash function acts as a random mapping, converting a series of successive counter values to a series of random numbers. Any function that performs a random mapping ought to be usable in a PRNG.

A block cipher, such as the Advanced Encryption Standard (AES) we saw in Chapters 5–7, is supposed to act as a random mapping. AES converts a 128-bit plaintext into a 128-bit ciphertext that looks like random gibberish. Lacking the key, it is infeasible to go backward from the ciphertext to the plaintext; there is no nonrandomness in AES's output that gives any clue about AES's input. This suggests that AES, or any block cipher, can be used as a PRNG's hash function.

Figure 31.4 shows how a block-cipher-based PRNG works. The PRNG is initialized by using the seed as the encryption key and setting a 64-bit counter (type `long` in Java) to 0. To generate a random number: increment the counter; set the least significant 64 bits of the plaintext block to the counter and set the remaining bits of the plaintext block to 0; encrypt the plaintext block yielding a ciphertext block; take the most significant 64 bits of the ciphertext block as a `long` value; and divide that by $2^{64}$ to get a double value in the range 0.0 to 1.0.

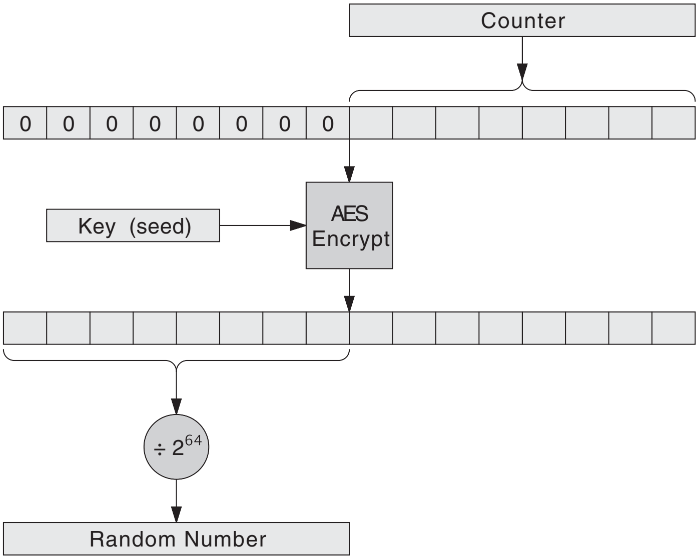


**Figure 31.4** Block-cipher-based PRNG using AES

While much slower than a typical PRNG (such as those described in Chapter 14), a block-cipher-based PRNG is much more secure. With a typical PRNG, if an adversary obtains some of the output random numbers, the adversary might be able to deduce the PRNG's internal state and thereby discover numbers the PRNG generated in the past, or discover numbers the PRNG will generate in the future. This would breach the security of an application that used the PRNG. With a block-cipher-based PRNG, such an attack is practically impossible. Determining the PRNG's internal state (encryption key or counter value) from the output random numbers would entail breaking the cipher. For this reason, secure applications often use a block-cipher-based PRNG to generate random numbers. Scientific applications, which don't need the security, use the simpler and faster PRNG algorithms.

A block-cipher-based PRNG might be secure, but is it random? To find out, we will perform the K-S test on a counter-mode PRNG using AES as the hash function. Because millions or billions of random numbers are required for a meaningful test, and because AES is slow compared to other PRNG hash functions, we want a program that can do the K-S test in parallel.

## 31.3 Sequential K-S Test Program

Here is the source code for class edu.rit.clu.monte.AesTestSeq. The command-line arguments are the encryption key (a 64-digit hexadecimal number) and *n*, the number of random numbers to generate. Use class edu.rit.clu.keysearch.MakeKey to create a key. The program prints *n*, the K-S statistic *D*, and the *p* value, as well as the running time.

```
$ java edu.rit.clu.monte.AesTestSeq $KEY 60000000
N = 60000000
D = 9.070410274467089E-5
P = 0.7068995841396919
66211 msec
```

```
package edu.rit.clu.monte;
import edu.rit.crypto.blockcipher.AES256Cipher;
import edu.rit.numeric.Statistics;
import edu.rit.pj.Comm;
import edu.rit.util.Hex;
import java.util.Arrays;
public class AesTestSeq
   {
   // Command line arguments.
   static byte[] key = new byte [32];
   static int N;

   // AES block cipher.
   static AES256Cipher cipher;

   // Plaintext and ciphertext blocks.
   static byte[] plaintext = new byte [16];
   static byte[] ciphertext = new byte [16];

   // Random data values.
   static double[] data;

   // 2^64.
   static double TWO_SUP_64;

   /**
    * Main program.
    */
   public static void main
      (String[] args)
      throws Exception
      {
      // Start timing.
      long time = -System.currentTimeMillis();

      // Validate command line arguments.
      if (args.length != 2) usage();
      Hex.toByteArray (args[0], key);
      N = Integer.parseInt (args[1]);

      // Set up AES block cipher.
      cipher = new AES256Cipher (key);

      // Allocate storage for random data values.
      data = new double [N];
```

```
         // Compute 2^64.
         TWO_SUP_64  = 2.0;         // 2^1
         TWO_SUP_64 *= TWO_SUP_64; // 2^2
         TWO_SUP_64 *= TWO_SUP_64; // 2^4
         TWO_SUP_64 *= TWO_SUP_64; // 2^8
         TWO_SUP_64 *= TWO_SUP_64; // 2^16
         TWO_SUP_64 *= TWO_SUP_64; // 2^32
         TWO_SUP_64 *= TWO_SUP_64; // 2^64
```

Here is step 1 of the statistical test procedure, to generate *n* random numbers. Because the K-S statistic requires the random numbers to be in ascending order, we have no choice except to store the random numbers we generate in an array and sort them.

```
         // Generate N random data values.
         for (int i = 0; i < N; ++ i)
            {
            longToBytes (i, plaintext, 8);
            cipher.encrypt (plaintext, ciphertext);
            data[i] = bytesToDouble (ciphertext, 0);
            }
```

Here is step 2 of the statistical test procedure, to compute the K-S statistic *D*. It saves time to store the random numbers in the data array as they are generated and sort them all at once, rather than store the numbers in the data array in sorted order as they are generated. The former can be done in $O(n \log n)$ time using Quicksort (implemented by `Arrays.sort()`). The latter is essentially an insertion sort, which requires $O(n^2)$ time.

```
         // Compute the K-S statistic, D.
         Arrays.sort (data);
         double N_double = N;
         double D = 0.0;
         double F_lower = 0.0;
         double F_upper;
         double x;
         for (int i = 0; i < N; ++ i)
            {
            F_upper = (i+1) / N_double;
            x = data[i];
            D = Math.max (D, Math.abs (x - F_lower));
            D = Math.max (D, Math.abs (x - F_upper));
            F_lower = F_upper;
            }
```

Here is step 3 of the statistical test procedure, to compute the *p* value. The static `ksPvalue()` method in class edu.rit.numeric.Statistics implements Equations 31.3–31.4.

```
       // Compute the p-value, P.
       double P = Statistics.ksPvalue (N, D);

       // Stop timing.
       time += System.currentTimeMillis();

       // Print results.
       System.out.println ("N = " + N);
       System.out.println ("D = " + D);
       System.out.println ("P = " + P);
       System.out.println (time + " msec”);
       }

    /**
     * Convert the given long value to eight bytes stored starting at
     * block[i].
     */
    private static void longToBytes
       (long value,
        byte[] block,
        int i)
       {
       for (int j = 7; j >= 0; — j)
          {
          block[i+j] = (byte) (value & 0xFF);
          value >>>= 8;
          }
       }

    /**
     * Convert the eight bytes starting at block[i] to a double
     * value.
     */
    private static double bytesToDouble
       (byte[] block,
        int i)
       {
       long result = 0L;
       for (int j = 0; j < 8; ++ j)
          {
```

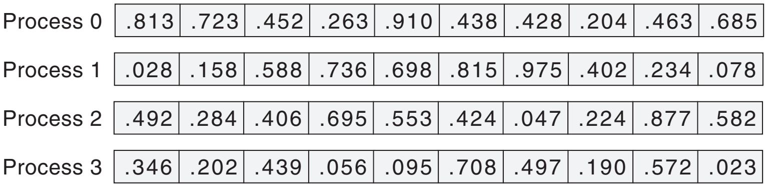
```
          result = (result << 8) | (block[i+j] & 0xFF);
          }
       return result / TWO_SUP_64 + 0.5;
       }
    }
```

Step 4 of the statistical test procedure, to decide whether the *p* value is too small or too large, is done by the user.

## 31.4 Parallel K-S Test Design

The K-S test program has three major time-consuming sections: generate *n* random numbers; sort them; and scan them to calculate *D*. Let's consider how to parallelize each section.

Generating *n* random numbers in parallel is straightforward. Instead of one process with an *n*-element array, we have *K* processes, each with an *n*/*K*-element array slice. We use type `long` for *n*, so we can scale up the program to more than two billion random numbers (the maximum possible with type `int`). We use a LongRange object to partition the index range 0 through *n*–1 into *K* subranges. Each process generates random numbers, starting with a counter value equal to the lower bound of the process's own subrange, and stores the random numbers in its own array slice. As an example, Figure 31.5 shows 40 random numbers partitioned among four processes.

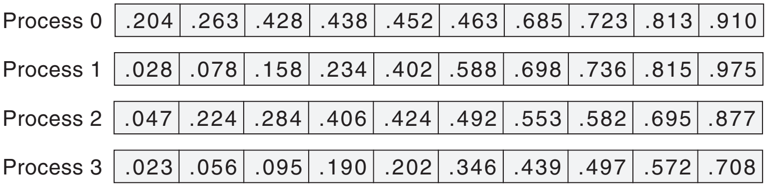


| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Process 0 | .813 | .723 | .452 | .263 | .910 | .438 | .428 | .204 | .463 | .685 |
| Process 1 | .028 | .158 | .588 | .736 | .698 | .815 | .975 | .402 | .234 | .078 |
| Process 2 | .492 | .284 | .406 | .695 | .553 | .424 | .047 | .224 | .877 | .582 |
| Process 3 | .346 | .202 | .439 | .056 | .095 | .708 | .497 | .190 | .572 | .023 |

**Figure 31.5** Random data generated in each process

Note the memory scalability of this design. We can scale up *n* as large as we please, as long as we also scale up *K* so that each process's array slice has at most about 268 million elements. The JVM puts a limit of $2^{31}$–1 bytes on the size of each object's or array's storage block, including storage for the object's fields or the array's elements, plus storage for JVM internal data about the object or array. With a `double` occupying 8 bytes, this translates into slightly fewer than $2^{28}$ elements in a `double` array. Of course, each node in the cluster must have enough physical memory to store the *n*/*K* array elements.

Next comes sorting the data. Figure 31.6 shows what happens if each process sorts its own slice.



| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Process 0 | .204 | .263 | .428 | .438 | .452 | .463 | .685 | .723 | .813 | .910 |
| Process 1 | .028 | .078 | .158 | .234 | .402 | .588 | .698 | .736 | .815 | .975 |
| Process 2 | .047 | .224 | .284 | .406 | .424 | .492 | .553 | .582 | .695 | .877 |
| Process 3 | .023 | .056 | .095 | .190 | .202 | .346 | .439 | .497 | .572 | .708 |

**Figure 31.6** Random data after sorting in each process

But this is not quite what we want. After sorting, we want the *entire* sorted array to be partitioned among the processes. That is, process 0 should have all the smallest numbers, process 1 should have all the next-smallest numbers, and so on, with the numbers divided evenly or close to evenly among the processes. Specifically, let's partition the random numbers $x$ in the range $0 \le x < 1$, such that process $k$ has all the random numbers in the range $k/K \le x < (k+1)/K$. In our example, process 0 has the range $0.00 \le x < 0.25$; process 1, $0.25 \le x < 0.50$; process 2, $0.50 \le x < 0.75$; and process 3, $0.75 \le x < 1.00$.

Because each process's slice is sorted, we can scan the slice from lowest to highest index to identify a sub-slice that is supposed to be located in each of the other processes, as shown in Figure 31.7.

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Process 0 | .204 | .263 | .428 | .438 | .452 | .463 | .685 | .723 | .813 | .910 |
| for→ | Pr. 0 | Pr. 1 | | | | | Pr. 2 | | Pr. 3 | |
| Process 1 | .028 | .078 | .158 | .234 | .402 | .588 | .698 | .736 | .815 | .975 |
| for→ | Pr. 0 | | | | Pr. 1 | Pr. 2 | | | Pr. 3 | |
| Process 2 | .047 | .224 | .284 | .406 | .424 | .492 | .553 | .582 | .695 | .877 |
| for→ | Pr. 0 | | Pr. 1 | | | | Pr. 2 | | | Pr. 3 |
| Process 3 | .023 | .056 | .095 | .190 | .202 | .346 | .439 | .497 | .572 | .708 |
| for→ | Pr. 0 | | | | | Pr. 1 | | | Pr. 2 | Pr. 3 |

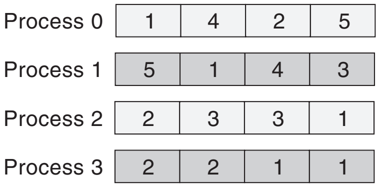
**Figure 31.7** Sub-slices to be located in each process

And we can identify the length of each sub-slice destined for each process, as shown in Figure 31.8.

| | | | | |
|---|---|---|---|---|
| Process 0 | 1 | 5 | 2 | 2 |
| Process 1 | 4 | 1 | 3 | 2 |
| Process 2 | 2 | 4 | 3 | 1 |
| Process 3 | 5 | 3 | 1 | 1 |

**Figure 31.8** Lengths of sub-slices to be located in each process

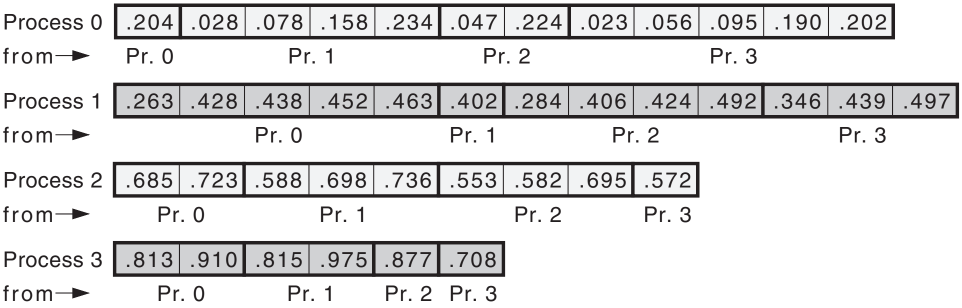
We want to send the sub-slices from process to process so that process 0 ends up with all the data in the range $0.00 \le x < 0.25$, process 1 ends up with all the data in the range $0.25 \le x < 0.50$, and so on. At this point every process knows the index range and length of the sub-slice the process will send to every other process. However, every process does not know the index range and length of the sub-slice the process will *receive from* every other process. To find this out, the processes do an **all-to-all** collective communication operation on the length information, yielding the lengths shown in Figure 31.9.



| | | | | |
|---|---|---|---|---|
| Process 0 | 1 | 4 | 2 | 5 |
| Process 1 | 5 | 1 | 4 | 3 |
| Process 2 | 2 | 3 | 3 | 1 |
| Process 3 | 2 | 2 | 1 | 1 |

**Figure 31.9** Lengths of sub-slices to be received from each process

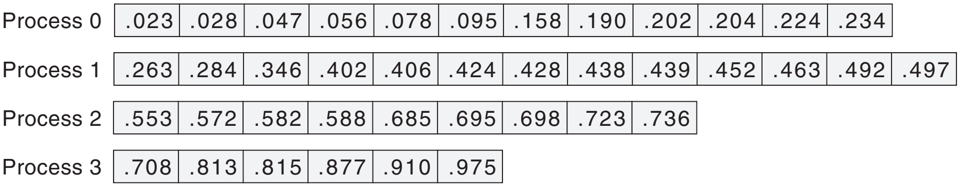
Each process adds up the sub-slice lengths to determine its total sorted slice length and allocates storage for its sorted slice. The processes then do another all-to-all operation to distribute the random numbers (Figure 31.10).



| | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Process 0 | .204 | .028 | .078 | .158 | .234 | .047 | .224 | .023 | .056 | .095 | .190 | .202 | |
| from→ | Pr. 0 | | Pr. 1 | | | | Pr. 2 | | | Pr. 3 | | | |
| Process 1 | .263 | .428 | .438 | .452 | .463 | .402 | .284 | .406 | .424 | .492 | .346 | .439 | .497 |
| from→ | | | Pr. 0 | | | Pr. 1 | | | Pr. 2 | | | Pr. 3 | |
| Process 2 | .685 | .723 | .588 | .698 | .736 | .553 | .582 | .695 | .572 | | | | |
| from→ | | Pr. 0 | | Pr. 1 | | | Pr. 2 | | Pr. 3 | | | | |
| Process 3 | .813 | .910 | .815 | .975 | .877 | .708 | | | | | | | |
| from→ | | Pr. 0 | | Pr. 1 | Pr. 2 | Pr. 3 | | | | | | | |

**Figure 31.10** Random data after all-to-all operation

And once each process sorts its incoming array, we have the sorted data partitioned among the processes the way we want it (Figure 31.11).



| | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Process 0 | .023 | .028 | .047 | .056 | .078 | .095 | .158 | .190 | .202 | .204 | .224 | .234 | |
| Process 1 | .263 | .284 | .346 | .402 | .406 | .424 | .428 | .438 | .439 | .452 | .463 | .492 | .497 |
| Process 2 | .553 | .572 | .582 | .588 | .685 | .695 | .698 | .723 | .736 | | | | |
| Process 3 | .708 | .813 | .815 | .877 | .910 | .975 | | | | | | | |

**Figure 31.11** Random data after final sort in each process

Every process will not necessarily end up with the same quantity of sorted data. But if the PRNG obeys a uniform probability distribution, and the program generates a large quantity of random numbers, the processes' array lengths should be nearly balanced.

Now that we have the random data sorted and partitioned among the processes, we can calculate *D* in parallel. In process 0, for the random number at index 0, the lower and upper values of the cumulative distribution function are 0/40 and 1/40; for index 1, 1/40 and 2/40; and so on. But in process 1, the random number at index 0 is not the first number in the cumulative distribution; it is the thirteenth. So in process 1, for the random number at index 0, the lower and upper values of the cumulative distribution function are 12/40 and 13/40; for index 1, 13/40 and 14/40; and so on. The cumulative distribution function starts at 0/40 in process 0, 12/40 in process 1, 25/40 in process 2, and 34/40 in process 3. Note the relation between the cumulative distribution function numerators and the slice lengths: the numerator in process *k* starts at the sum of the slice lengths in processes 0 through *k*–1. To determine these numerators, the processes do an **exclusive-scan** collective communication operation on the partitioned sorted slice lengths with sum as the reduction operator. A scan is necessary because the slice lengths might be different in every process.

Starting at the proper value for the cumulative distribution function, each process computes the maximum *D* for its own slice. In the example, the results are 0.066, 0.128, 0.137, and 0.167. The overall *D* value is then the maximum of the processes' individual *D* values, namely 0.167. The processes do a **reduce** collective communication operation, with maximum as the reduction operator, to put the overall *D* value into process 0. Process 0 then calculates the *p* value, 0.19459 in this example.

## 31.5 Parallel K-S Test Program

Here is the source code for the cluster parallel version, class edu.rit.clu.monte.AesTestSeq.

```
package edu.rit.clu.monte;
import edu.rit.crypto.blockcipher.AES256Cipher;
import edu.rit.mp.DoubleBuf;
import edu.rit.mp.IntegerBuf;
import edu.rit.mp.LongBuf;
import edu.rit.mp.buf.DoubleItemBuf;
import edu.rit.mp.buf.LongItemBuf;
import edu.rit.numeric.Statistics;
import edu.rit.pj.Comm;
import edu.rit.pj.reduction.DoubleOp;
import edu.rit.pj.reduction.LongOp;
import edu.rit.util.Hex;
import edu.rit.util.LongRange;
import edu.rit.util.Range;
import java.util.Arrays;
public class AesTestClu
   {
   // World communicator.
   static Comm world;
   static int size;
   static int rank;
   // Command line arguments.
   static byte[] key = new byte [32];
   static long N;

   // AES block cipher.
   static AES256Cipher cipher;

   // Plaintext and ciphertext blocks.
   static byte[] plaintext = new byte [16];
   static byte[] ciphertext = new byte [16];

   // Random data values, partitioned to be sent to all processes.
   static double[] sendData;

   // Number of data values sent from this process to each process,
   // plus total.
   static int[] sendLength;
   static int sendN;

   // Number of data values received by this process from each
   // process, plus total.
   static int[] recvLength;
   static int recvN;

   // Index ranges in the sendData array from which to obtain data
   // values sent to each process.
   static Range[] sendRanges;

   // Index ranges in the data array in which to store data values
   // received from each process.
   static Range[] recvRanges;

   // Random data values received by this process.
   static double[] data;

   // Number of data values in lower-ranked processes.
   static long lowerN;

   // 2^64.
   static double TWO_SUP_64;

   /**
    * Main program.
    */
   public static void main
      (String[] args)
      throws Exception
      {
      // Start timing.
      long time = -System.currentTimeMillis();

      // Initialize middleware.
      Comm.init (args);
      world = Comm.world();
      size = world.size();
      rank = world.rank();

      // Validate command line arguments.
      if (args.length != 2) usage();
      Hex.toByteArray (args[0], key);
      N = Long.parseLong (args[1]);

      // Set up AES block cipher.
      cipher = new AES256Cipher (key);

      // Compute 2^64.
      TWO_SUP_64  = 2.0;        // 2^1
      TWO_SUP_64 *= TWO_SUP_64; // 2^2
      TWO_SUP_64 *= TWO_SUP_64; // 2^4
      TWO_SUP_64 *= TWO_SUP_64; // 2^8
      TWO_SUP_64 *= TWO_SUP_64; // 2^16
      TWO_SUP_64 *= TWO_SUP_64; // 2^32
      TWO_SUP_64 *= TWO_SUP_64; // 2^64

      // Generate this process's subset of the N random data
      // values.
      LongRange indexRange =
         new LongRange(0,N-1).subrange (size, rank);
      long lb = indexRange.lb();
      long len = indexRange.length();
      sendData = new double [(int) len];
      for (long i = 0; i < len; ++ i)
         {
         longToBytes (lb+i, plaintext, 8);
         cipher.encrypt (plaintext, ciphertext);
         sendData[(int) i] = bytesToDouble (ciphertext, 0);
         }
      // If there's more than one process, do message passing.
      if (size > 1)
         {
         // Determine how many data values will be going to each
         // process.
         Arrays.sort (sendData);
         sendLength = new int [size];
         int prevj = 0;
         int j = 0;
         for (int i = 0; i < size; ++ i)
            {
            double threshold = ((double) (i+1)) / ((double) size);
            while (j < len && sendData[j] < threshold) ++ j;
            sendLength[i] = j - prevj;
            prevj = j;
            }

         // Determine how many data values will be coming from each
         // process.
         recvLength = new int [size];
         world.allToAll
            (IntegerBuf.sliceBuffers
               (sendLength, new Range(0,size-1).subranges (size)),
             IntegerBuf.sliceBuffers
               (recvLength, new Range(0,size-1).subranges (size)));

         // Transfer data values.
         sendRanges = new Range [size];
         sendN = 0;
         recvRanges = new Range [size];
         recvN = 0;
         for (int i = 0; i < size; ++ i)
            {
            sendRanges[i] = new Range(sendN,sendN+sendLength[i]-1);
            sendN += sendLength[i];
            recvRanges[i] = new Range(recvN,recvN+recvLength[i]-1);
            recvN += recvLength[i];
            }
         data = new double [recvN];
         world.allToAll
            (DoubleBuf.sliceBuffers (sendData, sendRanges),
             DoubleBuf.sliceBuffers (data, recvRanges));

         // Release storage for sent data values.
         sendData = null;
         // Determine how many data values ended up in lower-ranked
         // processes.
         LongItemBuf lowerNbuf = LongBuf.buffer (recvN);
         world.exclusiveScan (lowerNbuf, LongOp.SUM, 0L);
         lowerN = lowerNbuf.item;
         }

      // If there's only one process, don't bother with message
      // passing.
      else
         {
         data = sendData;
         sendData = null;
         recvN = (int) len;
         lowerN = 0;
         }

      // Compute the K-S statistic, D, for this process's random
      // data values.
      Arrays.sort (data);
      double N_double = N;
      double D = 0.0;
      double F_lower = lowerN / N_double;
      double F_upper;
      double x;
      for (int i = 0; i < recvN; ++ i)
         {
         F_upper = (lowerN+i+1) / N_double;
         x = data[i];
         D = Math.max (D, Math.abs (x - F_lower));
         D = Math.max (D, Math.abs (x - F_upper));
         F_lower = F_upper;
         }

      // Put the maximum of all processes' D values into process 0.
      DoubleItemBuf Dbuf = DoubleBuf.buffer (D);
      world.reduce (0, Dbuf, DoubleOp.MAXIMUM);
      D = Dbuf.item;

      // Compute the p-value, P.
      double P = Statistics.ksPvalue (N, D);

      // Stop timing.
      time += System.currentTimeMillis();
```

```
       // Print results in process 0.
       if (rank == 0)
          {
          System.out.println ("N = " + N);
          System.out.println ("D = " + D);
          System.out.println ("P = " + P);
          }
       System.out.println (time + " msec " + rank);
       }

    /**
     * Convert the given long value to eight bytes stored starting at
     * block[i]</TT>.
     */
    private static void longToBytes
       (long value,
        byte[] block,
        int i)
       {
       for (int j = 7; j >= 0; — j)
          {
          block[i+j] = (byte) (value & 0xFF);
          value >>>= 8;
          }
       }

    /**
     * Convert the eight bytes starting at block[i] to a double
     * value.
     */
    private static double bytesToDouble
       (byte[] block,
        int i)
       {
       long result = 0L;
       for (int j = 0; j < 8; ++ j)
          {
          result = (result << 8) | (block[i+j] & 0xFF);
          }
       return result / TWO_SUP_64 + 0.5;
       }
    }
```

## 31.6 Parallel K-S Test Program Performance

To collect running-time data for measuring the AesTestClu program's performance, we're going to do something a little different. Because the JVM puts a limit on the number of elements in an array, we can only go up to about $n$ = 268 million and still measure the program's running time on a single processor. We want to try problem sizes larger than that. So instead of collecting data for speedup, we will collect data for sizeup. As we go to more processors, we will also increase $n$ to keep the running time the same (assuming an ideal sizeup).

As $n$ scales up, the running time is dominated by the $O(n \log n)$-time sorting steps. Therefore, we will take the problem size $N$ to be $n \log_2 n$. Here are four sets of $n$ values, chosen so that $N(K) = K \cdot N(1)$. Because the "tardis" cluster nodes lack sufficient main memory to run more than one process with these problem sizes, we can scale up only to $K$=10 processes.

| $K$ | Set 1, $n$ | Set 2, $n$ | Set 3, $n$ | Set 4, $n$ |
|---:|---:|---:|---:|---:|
| 1 | 60M | 116M | 170M | 224M |
| 2 | 116M | 224M | 329M | 432M |
| 3 | 170M | 329M | 484M | 636M |
| 4 | 224M | 432M | 636M | 837M |
| 5 | 276M | 535M | 787M | 1035M |
| 6 | 329M | 636M | 936M | 1232M |
| 7 | 381M | 738M | 1084M | 1430M |
| 8 | 432M | 837M | 1232M | 1622M |
| 9 | 484M | 938M | 1378M | 1818M |
| 10 | 535M | 1035M | 1525M | 2007M |

Table 31.1 (at the end of the chapter) lists, and Figure 31.12 plots, the problem-size data for the AesTestClu program (derived from the running-time data in Table 31.2). The sizeups and sizeup efficiencies drop to 80–85 percent as we go to two processes because of the extra sorting and message passing the parallel version has to do. But, as the problem size and the number of processors continue to scale up, the sizeup efficiencies stay nearly constant, evincing good scalability.

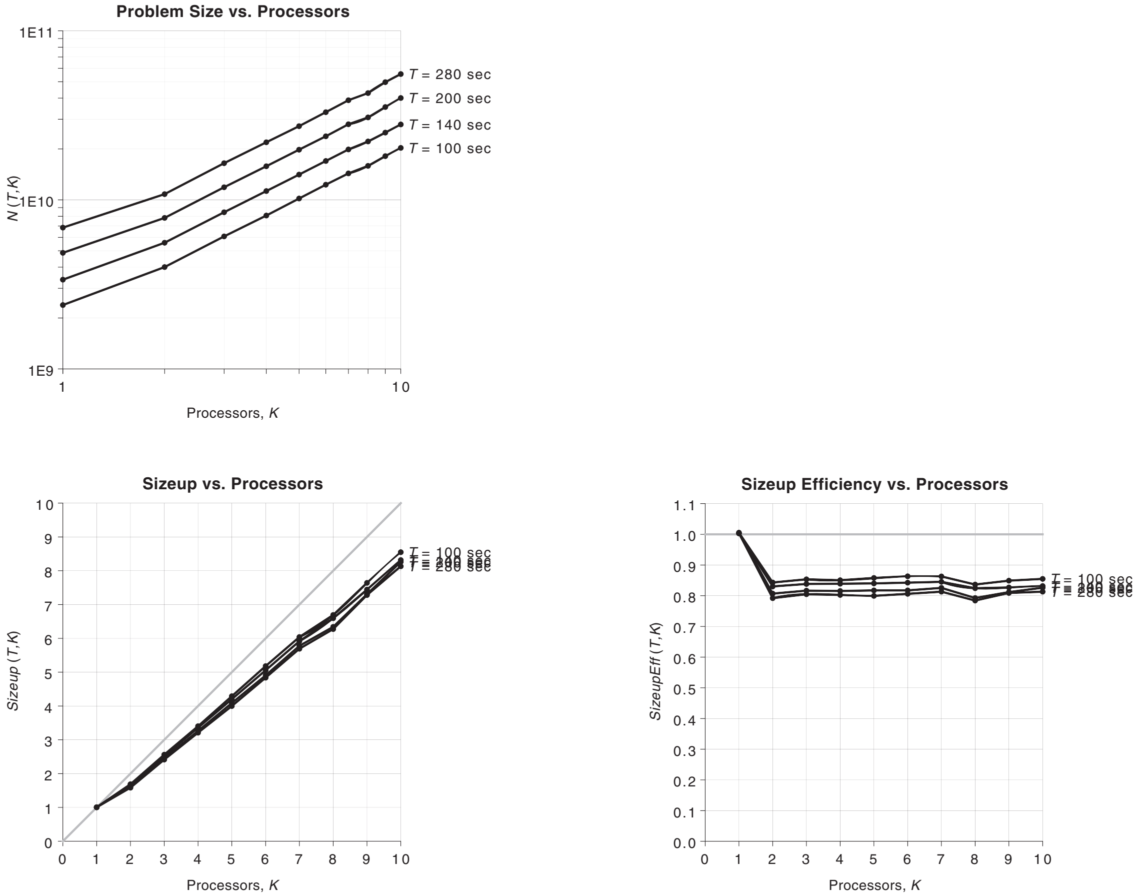


**Figure 31.12** AesTestSeq/AesTestClu problem-size metrics

Does the AES block-cipher-based PRNG pass the K-S test? Yes, it does. For the sample sizes just listed, the *p* values ranged from 0.11367 to 0.96589. An AES-based PRNG is secure *and* random (according to this one statistical test).

## 31.7 Collective Communication: All-to-All and Scan

In an all-to-all operation, every process sends one source buffer to every other process, and every process receives one destination buffer from every other process. In the first round of message passing, each process sends a message to the process one rank ahead and receives a message from the process one rank behind (Figure 31.13). In the second round of message passing, each process sends a message to the process two ranks ahead and receives a message from the process two ranks behind, and so on. Thus, the time required for an all-to-all is (*K*–1) times the time to send one message.

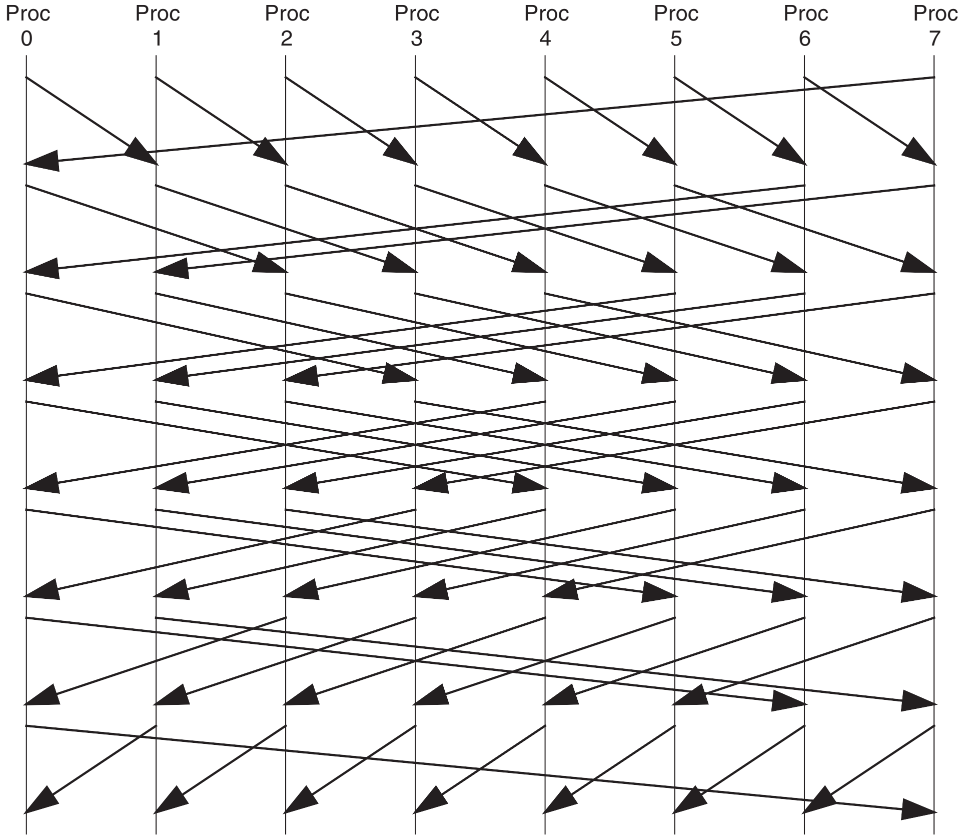


**Figure 31.13** All-to-all among eight processes

In a scan operation, the processes send messages to higher-ranked processes, which then combine the incoming data with the data in their buffers using the reduction operator (Figure 31.14). In the first round of message passing, each process sends a message to the process one rank ahead; in the second round, two ranks ahead; in the third round, four ranks ahead; and so on. Thus, the time required for a scan is ($\log_2 K$) times the time to send one message.

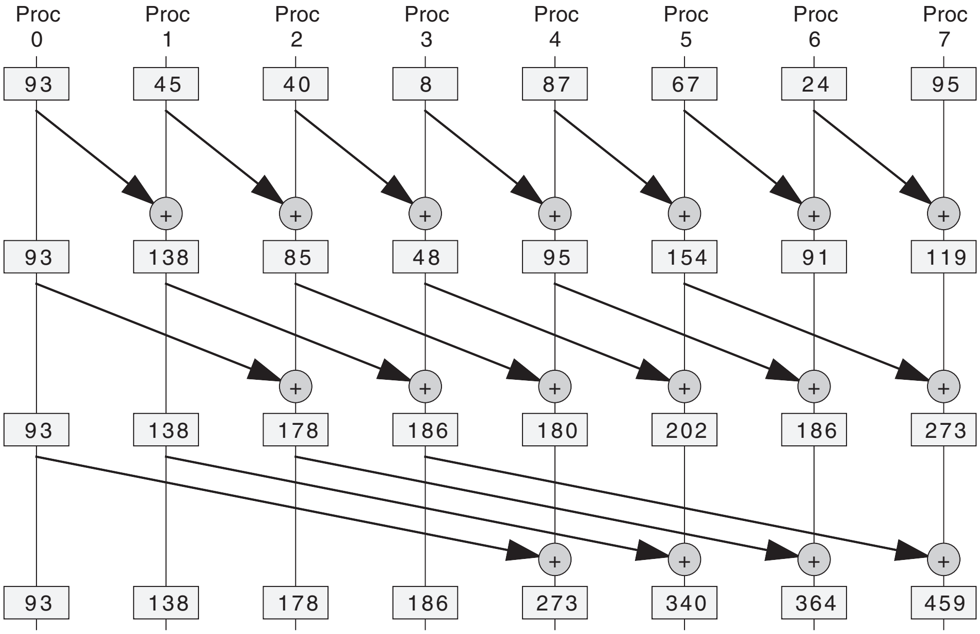


**Figure 31.14** Scan among eight processes, sum as the reduction operator

An exclusive-scan operation begins with each process sending its data to the process one rank ahead. Process 0 replaces the contents of its buffer with the given initial value. After that, an exclusive-scan is the same as a scan, except process 0 does not participate (Figure 31.15). Thus, the time required for an exclusive-scan is ($\log_2 K$)+1 times the time to send one message.

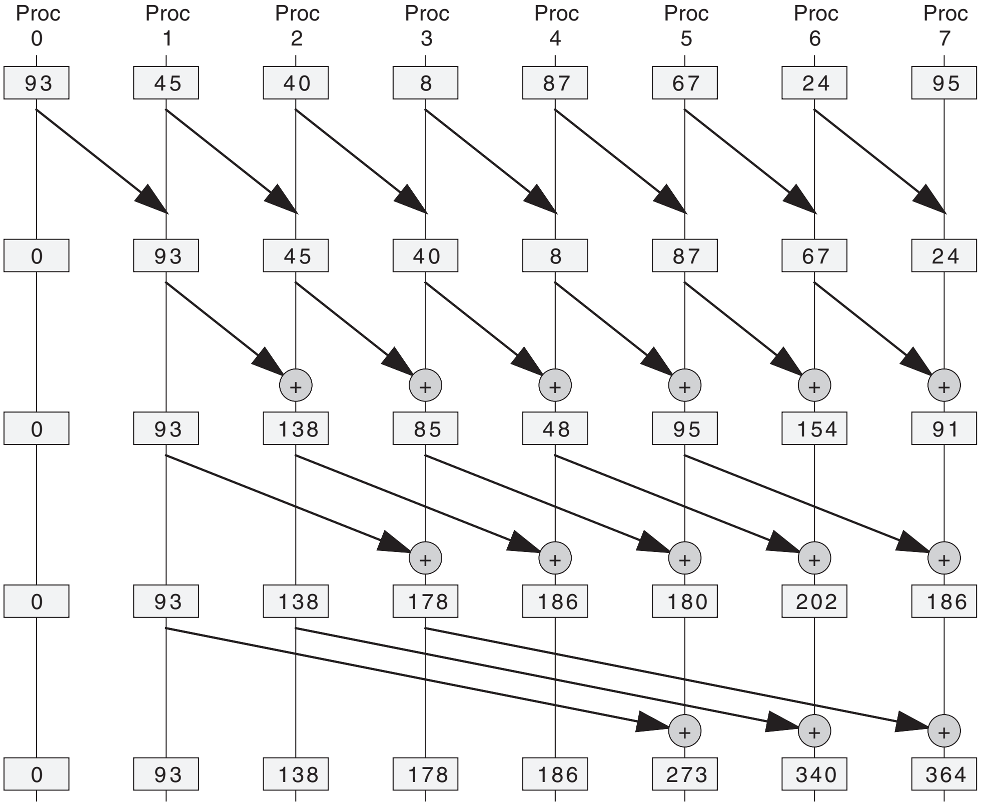


**Figure 31.15** Exclusive-scan among eight processes, sum as the reduction operator, initial value 0

Recapping the previous chapters, the collective communication operations fall into two categories: those that require $O(\log K)$ time—broadcast, flood, reduce, all-reduce, scan, exclusive-scan, and barrier; and those that require $O(K)$ time—scatter, gather, all-gather, and all-to-all—when implemented on a cluster parallel computer with one network interface on each node. Because of the fixed latency on each message, any program that uses collective communication operations eventually experiences a slowdown as the number of processors $K$ increases. The slowdown is more severe with the $O(K)$ operations than with the $O(\log K)$ operations. As we have seen in the previous chapters, a running-time model is essential for choosing the number of processors to get the maximum possible performance.

In Part II and Part III, we studied programming for SMP parallel computers and for cluster parallel computers. Now it's time to combine both sets of techniques and study programming for hybrid parallel computers—clusters of SMP machines—in Part IV.

## 31.8 For Further Information

On the K-S test:

- D. Knuth. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*, Third Edition. Addison-Wesley, 1998, Chapter 3.
- W. Press, S. Teukolsky, W. Vetterling, and B. Flannery. *Numerical Recipes: The Art of Scientific Computing, Third Edition.* Cambridge University Press, 2008, Chapter 14.

On cryptographically secure PRNGs:

- J. Kelsey, B. Schneier, and N. Ferguson. Yarrow-160: notes on the design and analysis of the Yarrow cryptographic pseudorandom number generator. In *Proceedings of the 6th Annual Workshop on Selected Areas in Cryptography*, 1999.
- N. Ferguson and B. Schneier. *Practical Cryptography*. Wiley Publishing, 2003, Chapter 10.

**Table 31.1** AesTestSeq/AesTestClu problem-size metrics

| *T* | *K* | *N* | *Sizeup* | *SzEff* | *T* | *K* | *N* | *Sizeup* | *SzEff* |
|---:|---:|---:|---:|---:|---:|---:|---:|---:|---:|
| 100000 | 0 | 2374144913 | | | 200000 | 0 | 4846968362 | | |
| 100000 | 1 | 2387102067 | 1.005 | 1.005 | 200000 | 1 | 4861143692 | 1.003 | 1.003 |
| 100000 | 2 | 4003334897 | 1.686 | 0.843 | 200000 | 2 | 7822263600 | 1.614 | 0.807 |
| 100000 | 3 | 6078956821 | 2.560 | 0.853 | 200000 | 3 | 11873458208 | 2.450 | 0.817 |
| 100000 | 4 | 8076651477 | 3.402 | 0.850 | 200000 | 4 | 15809249069 | 3.262 | 0.815 |
| 100000 | 5 | 10184523114 | 4.290 | 0.858 | 200000 | 5 | 19807528172 | 4.087 | 0.817 |
| 100000 | 6 | 12297315619 | 5.180 | 0.863 | 200000 | 6 | 23771639442 | 4.904 | 0.817 |
| 100000 | 7 | 14340403512 | 6.040 | 0.863 | 200000 | 7 | 28009266637 | 5.779 | 0.826 |
| 100000 | 8 | 15887837575 | 6.692 | 0.837 | 200000 | 8 | 30732768162 | 6.341 | 0.793 |
| 100000 | 9 | 18142668927 | 7.642 | 0.849 | 200000 | 9 | 35410687500 | 7.306 | 0.812 |
| 100000 | 10 | 20296395586 | 8.549 | 0.855 | 200000 | 10 | 40045459980 | 8.262 | 0.826 |
| 140000 | 0 | 3358824912 | | | 280000 | 0 | 6825397549 | | |
| 140000 | 1 | 3372463372 | 1.004 | 1.004 | 280000 | 1 | 6852090005 | 1.004 | 1.004 |
| 140000 | 2 | 5577652131 | 1.661 | 0.830 | 280000 | 2 | 10811994174 | 1.584 | 0.792 |
| 140000 | 3 | 8447269461 | 2.515 | 0.838 | 280000 | 3 | 16481614874 | 2.415 | 0.805 |
| 140000 | 4 | 11273790241 | 3.356 | 0.839 | 280000 | 4 | 21911340436 | 3.210 | 0.803 |
| 140000 | 5 | 14109658443 | 4.201 | 0.840 | 280000 | 5 | 27286609188 | 3.998 | 0.800 |
| 140000 | 6 | 16980408785 | 5.055 | 0.843 | 280000 | 6 | 33013374000 | 4.837 | 0.806 |
| 140000 | 7 | 19873081771 | 5.917 | 0.845 | 280000 | 7 | 38852280313 | 5.692 | 0.813 |
| 140000 | 8 | 22152637595 | 6.595 | 0.824 | 280000 | 8 | 42823913752 | 6.274 | 0.784 |
| 140000 | 9 | 24997519962 | 7.442 | 0.827 | 280000 | 9 | 49700039351 | 7.282 | 0.809 |
| 140000 | 10 | 27931394043 | 8.316 | 0.832 | 280000 | 10 | 55502306729 | 8.132 | 0.813 |

**Table 31.2** AesTestSeq/AesTestClu running-time metrics

| *K* | *T* | *N* | *K* | *T* | *N* | *K* | *T* | *N* |
|---:|---:|---:|---:|---:|---:|---:|---:|---:|
| seq | 66443 | 1550307550 | 4 | 76690 | 6213518862 | 8 | 77683 | 12392549023 |
| seq | 129875 | 3107587749 | 4 | 153997 | 12392549023 | 8 | 156962 | 24809226044 |
| seq | 191953 | 4647963116 | 4 | 237568 | 18599471170 | 8 | 247020 | 37204373496 |
| seq | 255258 | 6213518862 | 4 | 317128 | 24809226044 | 8 | 319916 | 49625295464 |
| 1 | 65948 | 1550307550 | 5 | 75079 | 7739065675 | 9 | 75614 | 13963608994 |
| 1 | 129319 | 3107587749 | 5 | 154294 | 15512305553 | 9 | 157270 | 27957101896 |
| 1 | 191434 | 4647963116 | 5 | 236709 | 23257257467 | 9 | 236835 | 41835981806 |
| 1 | 254341 | 6213518862 | 5 | 319844 | 30995128049 | 9 | 314147 | 55921143787 |
| 2 | 77241 | 3107587749 | 6 | 74472 | 9308565560 | 10 | 74936 | 15512305553 |
| 2 | 156156 | 6213518862 | 6 | 153829 | 18599471170 | 10 | 156051 | 30995128049 |
| 2 | 240507 | 9308565560 | 6 | 236805 | 27894609558 | 10 | 231450 | 46521897197 |
| 2 | 321519 | 12392549023 | 6 | 315366 | 37204373496 | 10 | 315242 | 62021103696 |
| 3 | 75831 | 4647963116 | 7 | 74841 | 10860487203 | | | |
| 3 | 154547 | 9308565560 | 7 | 153503 | 21740775635 | | | |
| 3 | 237040 | 13963608994 | 7 | 233569 | 32534869890 | | | |
| 3 | 316133 | 18599471170 | 7 | 314094 | 43491116241 | | | |

# PART III

# Exercises

1. Change the Program1Clu program in Chapter 19 so the start and finish times always print in rank order, for example, as in the following:

```
$ java -Dpj.np=4 Program1Clu
Job 13, thug05, thug06, thug07, thug08
x = 1 call start = 25 msec
x = 2 call start = 54 msec
x = 3 call start = 34 msec
x = 4 call start = 60 msec
x = 1 call finish = 1219 msec
x = 2 call finish = 1219 msec
x = 3 call finish = 1198 msec
x = 4 call finish = 1226 msec
```

2. Run the original and the modified Program1Clu programs on your cluster parallel computer. Compare the running times, speedups, and efficiencies of the two versions. What is causing the discrepancy, if any?

*Exercises 3–4*. Here is a cluster parallel program written using Parallel Java.

```
public class Foo
   {
   public static void main
      (String[] args)
      throws Exception
      {
      Comm.init (args);
      Comm world = Comm.world();
      int size = world.size();
      int rank = world.rank();
      IntegerItemBuf buf = IntegerBuf.buffer();
      buf.item = rank+1;
      world.reduce (0, buf, IntegerOp.PRODUCT);
      if (rank == 0) System.out.println (buf.item);
      }
   }
```

3. What does the program print when run on eight processes?

4. In general, what function of *K,* the number of processes, does the program compute? Explain how the program computes this function.

5. A cluster parallel program runs on four processes and does the following. The program gets a double-precision floating-point number *x* from the first command-line argument and calculates four mathematical functions of *x,* namely sin *x,* cos *x,* tan *x,* and ln *x.* Each process computes a different function of *x* in parallel. The program prints four lines on the standard output with the computed values of sin *x,* cos *x,* tan *x,* and ln *x,* one per line, in that order. The program does no error handling. Write the Parallel Java code for this program.

*Exercises 6–10.* A cluster parallel program has six data items of type `long` that must be transferred from process 0 to process 1. Write Parallel Java code fragments for process 0 and process 1 to create the communication buffers and transfer the data for each of the following scenarios:

6. The data items are in separate local variables of the `main()` method.

7. The data items are the elements of an array.

8. The data items are the fields of a serializable object.

9. Discuss the pros and cons of the preceding three scenarios regarding coding effort.

10. Discuss the pros and cons of the preceding three scenarios regarding message send time.

*Exercises 11–12.* A cluster parallel program consisting of 12 processes broadcasts a message from root process 2 to all the processes. The time to send the message from one process to one other process is *T.*

11. How much time does it take to broadcast the message?

12. Draw a diagram showing the pattern of point-to-point messages that are sent to broadcast the message to all processes. Show the processes along the horizontal axis and time increasing downward along the vertical axis.

*Exercises 13–14.* A cluster parallel program consists of eight processes. Each process has a buffer containing an integer. In processes 0 through 7, the buffers contain the values 18, 19, 95, 19, 66, 99, 41, and 75, respectively. A reduction is performed into process 0, using addition as the reduction operator. The time to send a message containing one integer is 4 milliseconds. The time to perform the reduction operator is negligible.

13. How much time does it take to do the reduction?
14. Draw a diagram showing the pattern of point-to-point messages that are sent to perform the reduction. Show the processes along the horizontal axis and time increasing downward along the vertical axis. Show the value that is sent in each message and the final value stored in process 0's buffer.

*Exercises 15–16.* A cluster parallel program consists of 16 processes. Each process has a buffer containing an integer. In processes 0 through 15, the buffers contain the values 7, 4, 31, 90, 18, 9, 95, 58, 64, 92, 20, 46, 78, 8, 59, and 3, respectively. An all-reduce operation is performed, using addition as the reduction operator. The time to send a message containing one integer is 4 milliseconds. The time to perform the reduction operator is negligible.

15. How much time does it take to do the all-reduce?
16. Draw a diagram showing the pattern of point-to-point messages that are sent to perform the all-reduce. Show the processes along the horizontal axis and time increasing downward along the vertical axis. Show the value that is sent in each message and the final value stored in each process's buffer.

*Exercises 17–18.* A cluster parallel program consists of 16 processes. Each process has a buffer containing an integer. In processes 0 through 15, the buffers contain the values 7, 4, 31, 90, 18, 9, 95, 58, 64, 92, 20, 46, 78, 8, 59, and 3, respectively. A scan operation is performed, using addition as the reduction operator. The time to send a message containing one integer is 4 milliseconds. The time to perform the reduction operator is negligible.

17. How much time does it take to do the scan?
18. Draw a diagram showing the pattern of point-to-point messages that are sent to perform the scan. Show the processes along the horizontal axis and time increasing downward along the vertical axis. Show the value that is sent in each message and the final value stored in each process's buffer.

*Exercises 19–20.* A cluster parallel program consists of 16 processes. Each process has a buffer containing an integer. In processes 0 through 15, the buffers contain the values 7, 4, 31, 90, 18, 9, 95, 58, 64, 92, 20, 46, 78, 8, 59, and 3, respectively. An exclusive-scan operation is performed, using addition as the reduction operator. The time to send a message containing one integer is 4 milliseconds. The time to perform the reduction operator is negligible.

19. How much time does it take to do the exclusive-scan?
20. Draw a diagram showing the pattern of point-to-point messages that are sent to perform the exclusive-scan. Show the processes along the horizontal axis

and time increasing downward along the vertical axis. Show the value that is sent in each message and the final value stored in each process's buffer.

*Exercises 21–26.* Here is a cluster parallel version of Floyd's Algorithm for computing all shortest paths in an $n$-vertex graph whose distance matrix is $d$, an $n$-by-$n$ matrix. Each element in $d$ is a Java double-precision floating-point number. This version uses the scatter-gather pattern rather than the parallel input/output files pattern. The sequential version of Floyd's Algorithm is the same, without the communication operations.

```
1  Scatter row slices of D from process 0 to each process
2  For i in 0 .. N-1
3      Broadcast row i of D to all processes
4      For r in this process's subrange of 0 .. N-1
5          For c in 0 .. N-1
6              D[r,c] = min (D[r,c], D[r,i] + D[i,c])
7  Gather row slices of D from each process into process 0
```

$d$ is divided into equal-sized row slices. Measurements show that executing the statement on line 6 takes 0.01 microseconds. Also, measurements show that sending a message from one process to another takes $(400 + 0.8B)$ microseconds, where $B$ is the number of bits of data in the message.

21. Give an expression for the running time $T_1$ in microseconds of the sequential version of Floyd's Algorithm as a function of $n$, the number of nodes. Ignore the loop overhead time.

22. Give an expression for the time $T_2$ in microseconds needed to send all the messages for the scatter operation on line 1 as a function of $n$, the number of nodes, and $K$, the number of parallel processes.

23. Give an expression for the time $T_3$ in microseconds needed to send all the messages for the broadcast operation on line 3 as a function of $n$, the number of nodes, and $K$, the number of parallel processes.

24. Give an expression for the running time $T_4$ in microseconds of the cluster parallel version of Floyd's Algorithm as a function of $n$, the number of nodes, and $K$, the number of parallel processes. Ignore the loop overhead time.

25. How many parallel processes should be used to obtain the smallest running time for a 1,000-vertex graph? (Note that the number of parallel processes must be an integer.)

26. What is the largest speedup that can be obtained for a 1000-vertex graph?

*Exercises 27–29.* A two-dimensional **grayscale image** is stored in a file. Each pixel's value is an integer from 0 (white) to 255 (black). Each pixel value is stored in one byte of the file. The pixels are stored in the following order: the first row of pixels from left to right, then the second row of pixels from left to right, and so on. An image-processing program performs a **thresholding** operation on the image. The program calculates $M$, the average of all the pixel values. Then, the program changes each pixel as follows: if the pixel value is less than or equal to $M$, the pixel is set to white; otherwise, the pixel is set to black. The program reads the image from a given input file, performs the thresholding operation, and writes the result into a given output file. The program runs on a cluster parallel computer to get a speedup when processing large images. Process 0 is the only process allowed to read or write files.

27. Describe how the image should be partitioned among the parallel processes to achieve a balanced load.
28. Give a pseudocode description of the cluster parallel program. Be especially clear in your description of any message passing operations needed. Describe what the message passing operations are; describe from which process and from which data structure the data comes; and describe into which process and into which data structure the data goes.
29. Considering only computation and communication (omitting file I/O), describe how close to the ideal speedup you expect the parallel program to achieve as the number of parallel processes increases.

*Exercises 30–34.* In the image-processing cluster parallel program from Exercises 27–29, it takes 0.4 microseconds per pixel to calculate the average pixel value, and it takes 0.5 microseconds per pixel to compare the pixels to the average and set the pixels to white or black. Sending a message takes $(200 + 0.01B)$ microseconds, where $B$ is the number of bits of data in the message. The image contains $n$ rows and $n$ columns of pixels. Assume $n$ is a large number. The parallel program runs in $K$ parallel processes.

30. Give an expression in terms of $n$ for $T_1$, the running time of a sequential version of the program. Consider only computation (omit file I/O). You may leave certain terms out of the expression if they have a negligible effect on the running time; if so, give a justification for omitting these terms.
31. Give an expression in terms of $n$ and $K$ for $T_2$, the time needed to do all the communication in the parallel program (omit file I/O). You may leave certain terms out of the expression if they have a negligible effect on the running time; if so, give a justification for omitting these terms.
32. Give an expression in terms of $n$ and $K$ for $T_3$, the running time of the parallel program. Consider only computation and communication (omit file I/O). You may leave certain terms out of the expression if they have a negligible effect on the running time; if so, give a justification for omitting these terms.
33. Give an expression in terms of $n$ for the number of processes that would yield the best performance for the parallel program.
34. For a 1000 × 1000-pixel image, how many processes should be used to get the best performance, and what would the speedup be for this number of processes? (Note that the number of parallel processes must be an integer.)

*Exercises 35–36.* Alice, the biologist, does an experiment to measure the mutation rate of a certain species of bacterium. She takes one bacterium and clones it. From one clone, she extracts the DNA and determines the bacterium's complete genome (DNA sequence). She puts the other clone in a Petri dish, where it starts to multiply. Once an hour, Alice takes a number of bacteria out of the Petri dish, extracts their DNA, and determines each bacterium's complete genome. At the end of her experiment, Alice has a large number of DNA sequence files; a DNA sequence is just a very long character string. One file contains the original bacterium's genome. Every other file contains one descendant bacterium's genome; the time (number of hours since the start of the experiment) at which the bacterium was sampled is also recorded in the file. Alice needs to analyze her files to produce the following table:

| *Time* | *Mutations* |
|---|---|
| 1 | 1.14 |
| 2 | 2.30 |
| 3 | 3.97 |
| 4 | 8.75 |
| ⋮ | ⋮ |

where *Time* is the number of hours since the start of the experiment and *Mutations* is the average number of mutations per bacterium sampled at that time. Each position at which a sampled bacterium's DNA sequence differs from the original bacterium's DNA sequence counts as one mutation. All the DNA sequences are the same length.

You are designing a cluster parallel program to analyze Alice's DNA sequence files and produce the preceding table. Assume that any process can read any file directly.

35. Describe how the computation should be partitioned among the parallel processes to achieve a balanced load.

36. Give a pseudocode description of the cluster parallel program. The pseudocode must show the variables the program uses and the processing steps the program executes. Be especially clear in your description of any message passing operations needed. Describe what the message passing operations are, describe from which process and from which data structure the data comes, and describe into which process and into which data structure the data goes.

*Exercises 37–40.* In the cluster parallel program from Exercises 35–36, it takes 0.01 microseconds per character position to calculate the number of mutations in a DNA sequence. Sending a message takes $(200 + 0.01B)$ microseconds, where $B$ is the number of bits of data in the message. Every DNA sequence is $L$ characters long. There are $N$ DNA sequences (not counting the original bacterium's DNA sequence). There are $S$ sample times (each sample time occurring some number of hours after the start of the experiment). The program runs in $K$ parallel processes.

37. Give an expression for the running time of a sequential version of the program in terms of the variables $L$, $N$, $S$, and $K$ (whichever variables are necessary). Consider only computation (omit file I/O). You may leave certain terms out of

the expression if they have a negligible effect on the running time; if so, give a justification for omitting these terms.

38. Give an expression for the time needed to do all the communication in the parallel program (omit file I/O) in terms of the variables $L$, $N$, $S$, and $K$ (whichever variables are necessary). Consider only computation (omit file I/O). You may leave certain terms out of the expression if they have a negligible effect on the running time; if so, give a justification for omitting these terms.

39. Give an expression for the running time of the parallel program in terms of the variables $L$, $N$, $S$, and $K$ (whichever variables are necessary). Consider only computation (omit file I/O). You may leave certain terms out of the expression if they have a negligible effect on the running time; if so, give a justification for omitting these terms.

40. Alice runs her experiment for 24 hours. Each hour she samples 5 bacteria. Each bacterium's DNA sequence is 10,000 characters long. She can analyze her data on a 32-processor cluster or on a 64-processor cluster. Which cluster should she use?

*Exercises 41–44.* Given an integer $i > 0$, consider the following procedure:

```
$x \leftarrow i$
While $x > 1$:
    If $x$ is even:
        $x \leftarrow x/2$
    Else:
        $x \leftarrow 3x+1$
```

The **Collatz Conjecture**, proposed by Lothar Collatz in 1937, states that for every $i > 0$, the preceding procedure terminates; that is, $x$ eventually becomes 1. While mathematicians believe the Collatz Conjecture is true, no one has been able to prove it.

41. Write a sequential program to investigate whether the Collatz Conjecture is true for all values of $i$ from 1 through $N$, where $N$ is a command-line argument. Use type `long` so $N$ can be as large as $2^{63}–1$. The program also has a command-line argument *MaxIter* (type `long`). In the preceding procedure, for a certain value of $i$, if $x$ reaches 1 before the number of while loop iterations reaches *MaxIter*, then the Collatz Conjecture is true for $i$. If the number of while loop iterations reaches *MaxIter* before $x$ reaches 1, then the Collatz Conjecture *may* be false for $i$. (The Collatz Conjecture is not *definitely* false for $i$ because $x$ might reach 1 with further iterations, but the program has to stop somewhere.) The program prints the values of $i$ for which the Collatz Conjecture may be false.

42. Describe the sequential dependencies, if any, in the program. Is it possible to parallelize the program?

43. If possible, write a cluster parallel program to investigate the Collatz Conjecture. The parallel program has the same command-line arguments and the same output as the sequential program. Measure the parallel program's running times as a function of $N$ and $K$, calculate the program's running-time metrics, and improve the program's design if necessary.
44. Do you have to do anything to achieve load balance in the parallel program? If so, describe how to balance the load. If not, explain why not.

*Exercises 45–49.* A **three-dimensional random walk** is defined as follows. A particle is initially positioned at (0, 0, 0) in the X-Y-Z coordinate space. The particle does a sequence of $N$ steps. At each step, the particle chooses one of the six directions left, right, ahead, back, up, or down at random, then moves one unit in that direction. Specifically, if the particle is at $(x, y, z)$:

With probability 1/6 the particle moves left to $(x–1, y, z)$.
With probability 1/6 the particle moves right to $(x+1, y, z)$.
With probability 1/6 the particle moves back to $(x, y–1, z)$.
With probability 1/6 the particle moves ahead to $(x, y+1, z)$.
With probability 1/6 the particle moves down to $(x, y, z–1)$.
With probability 1/6 the particle moves up to $(x, y, z+1)$.

45. Write a sequential program to calculate the particle's final position. The program's command-line arguments are the random seed and the number of steps $N$. The program prints the particle's final position $(x, y, z)$ as well as the particle's final distance from the origin.
46. Describe the sequential dependencies, if any, in the program. Is it possible to parallelize the program?
47. If possible, write a cluster parallel program to calculate the particle's final position. The parallel program has the same command-line arguments and the same output as the sequential program. Measure the parallel program's running times as a function of $N$ and $K$, calculate the program's running time metrics, and improve the program's design if necessary.
48. What is the particle's expected final distance from the origin as a function of the number of steps $N$?
49. Run your program for a large number of steps and a variety of different random seeds. Do the particle's computed final distances from the origin agree with the expected final distance?

*Exercises 50–54.* Measurements on the sequential heat distribution program in Chapter 30 (class edu.rit.clu.heat.HotSpotSeq) running on the "tardis" cluster show that the running time of the calculation section $T_{\text{calc}}$, and the number of iterations, for an $n \times n$-element mesh for various mesh sizes are the following:

| $n$ | $T_{calc}$ (msec) | *Iterations* |
|---|---|---|
| 1260 | 60950 | 2539 |
| 1590 | 122652 | 3233 |
| 2000 | 266850 | 4099 |
| 2520 | 564505 | 5204 |
| 3180 | 993789 | 6618 |
| 4000 | 2048529 | 8388 |

50. Derive a formula for the calculation time for one iteration as a function of $n$, the mesh size.

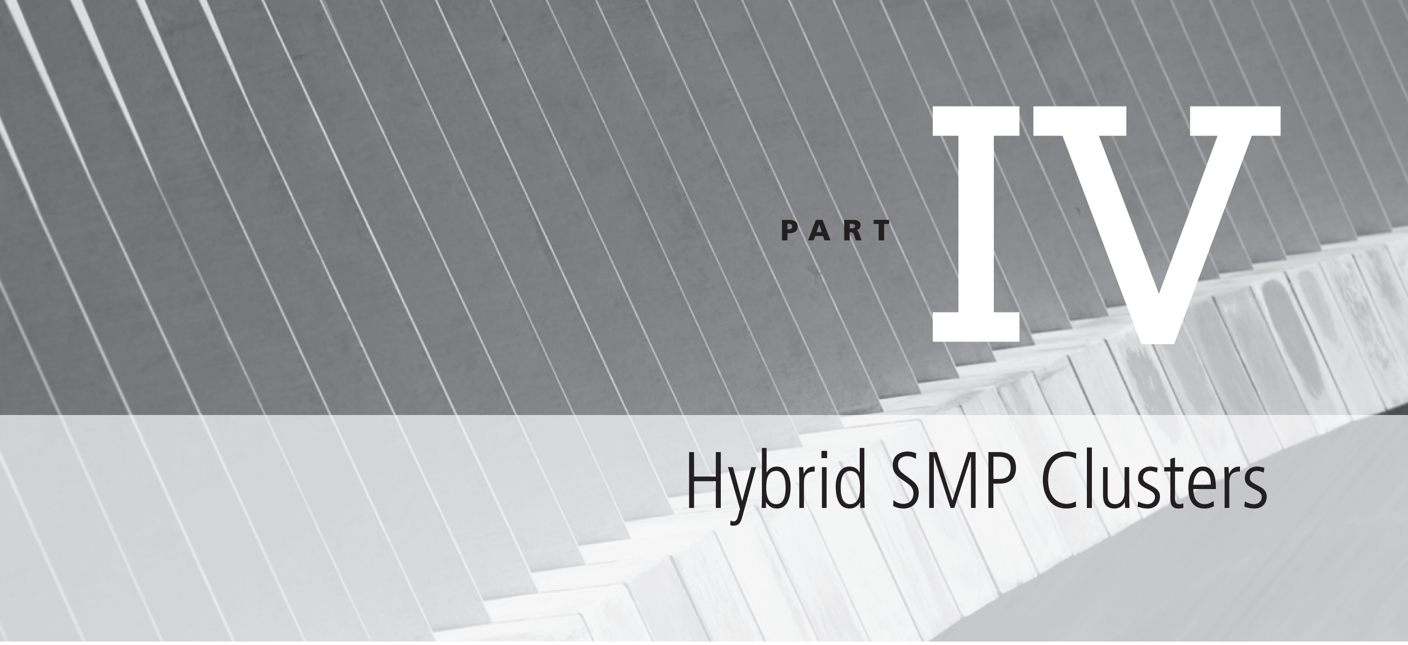
51. For the parallel version of the program, derive a formula for the communication time for one iteration as a function of $n$, the mesh size, and $K$, the number of processors. Use the message send time model for the "tardis" cluster. Assume each process runs on a different processor.

52. Taking into account the calculation and the communication, derive formulas for the speedup and efficiency of the parallel version as a function of $n$, the mesh size, and $K$, the number of processors.

53. What does the preceding model tell you about the program's scalability?

54. Compare the speedups and efficiencies predicted by the preceding model with those measured in Table 30.1. What is causing the discrepancy if any? (*Hint:* The running times in Table 30.1 are for the entire program.)

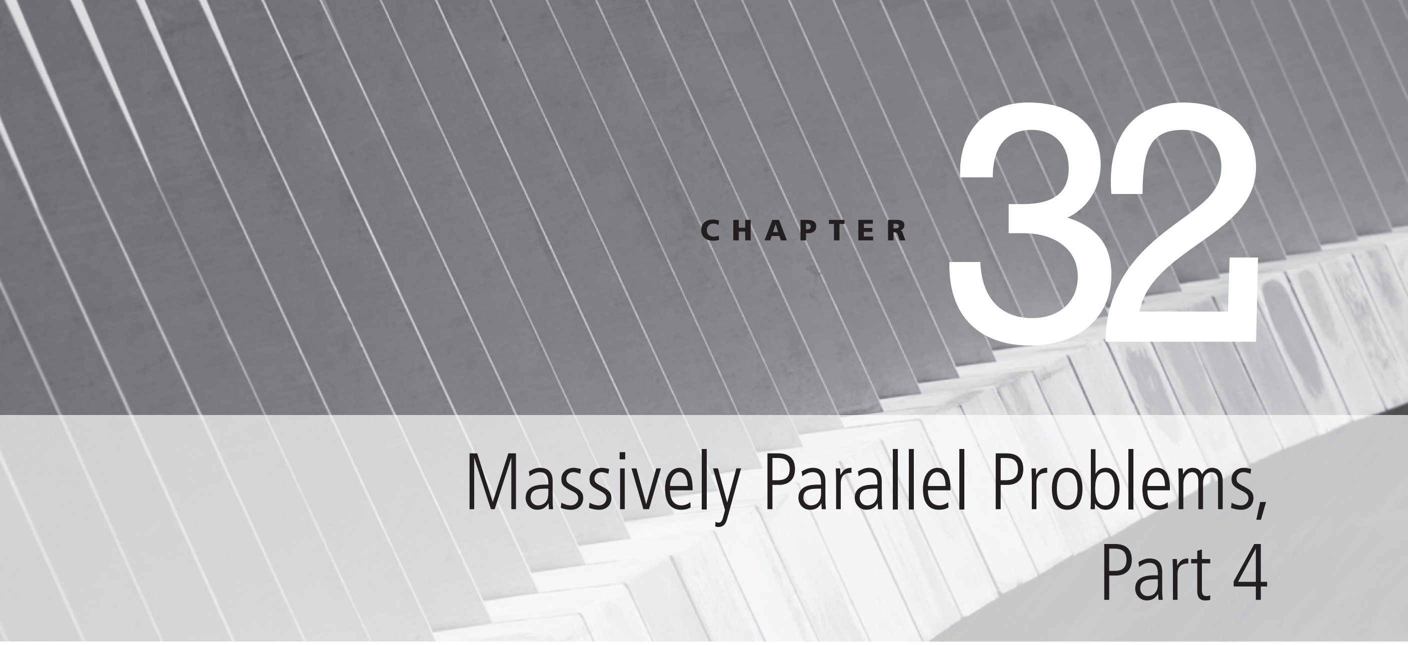
*This page intentionally left blank*



# PART IV

# Hybrid SMP Clusters

*This page intentionally left blank*



# CHAPTER 32

# Massively Parallel Problems, Part 4

in which we learn how to use both SMP parallel programming techniques and cluster parallel programming techniques to develop hybrid parallel programs; and we implement a massively parallel cryptographic problem one last time as a hybrid parallel program

# CHAPTER 32 Massively Parallel Problems, Part 4

## 32.1 Hybrid Parallel Program Design

To a certain extent, the SMP parallel programming techniques we studied in Part II and the cluster parallel programming techniques we studied in Part III have been just a prelude to *real* parallel programming. As mentioned in Chapter 2, an SMP parallel program can scale up only so far before hitting limits on a single SMP parallel computer—limited memory, limited number of CPUs. Scaling up to larger problem sizes or more processors requires a cluster parallel computer. Nowadays, the individual nodes of a cluster parallel computer are likely to be SMP (multicore) machines; single-core machines are becoming hard to find. To take full advantage of a hybrid parallel computer—a cluster of SMPs—requires using both SMP and cluster parallel programming techniques in the same program: message passing between nodes; shared memory and multithreading within each node. We've already got the techniques; now we just need to see how to combine them.

As we did in Part II and Part III, we'll begin our study of hybrid parallel programming with the simplest kind of parallel program, a **massively parallel program**. As our example, we'll revisit the AES partial key search program from Chapter 7 (SMP version) and Chapter 21 (cluster version). Recall that the program takes four arguments: a 128-bit plaintext block; a 128-bit ciphertext block; a 256-bit key with a certain number of low-order bits missing; and $n$, the number of missing key bits. The ciphertext was produced by encrypting the plaintext using the (complete) key. The program's job is to find the correct key by trying to encrypt the plaintext using all $2^n$ possible keys. The program prints the key that successfully reconstructs the ciphertext.

The sequential version of the program tried every possible value of the least significant key bits from 0 to $2^n–1$ in a regular loop.

```
for (int cntr = 0; cntr < maxcounter; ++ cntr)
   {
   // Try key
   }
```

The SMP parallel version of the program used a parallel loop to try every possible value of the least significant key bits from 0 to $2^n–1$.

```
new ParallelTeam().execute (new ParallelRegion()
   {
   public void run() throws Exception
```

```
         {
         execute (0, maxcounter, new IntegerForLoop()
            {
            public void run (int first, int last)
               {
               for (int cntr = first; cntr <= last; ++ cntr)
                  {
                  // Try key
                  }
               }
            });
         }
      });
```

The parallel loop partitioned the keys equally among the parallel team threads. For example, with $n = 10$ missing key bits, $N = 2^n = 1{,}024$ keys to try, and $K = 4$ threads, here are the keys each thread tried:

| Thread | Keys |
|---|---|
| 0 | 0 – 255 |
| 1 | 256 – 511 |
| 2 | 512 – 767 |
| 3 | 768 – 1,023 |

In the cluster parallel version, each process used a Range object to partition the range of key values into equal subranges and to pick the subrange corresponding to the process's rank. Each process then used a regular loop to try every key from the lower bound to the upper bound of that subrange.

```
Range chunk = new Range(0,maxcounter-1) .subrange(size,rank);
int lb = chunk.lb();
int ub = chunk.ub();
for (int cntr = lb; cntr <= ub; ++ cntr)
   {
   // Try key
   }
```

The `subrange()` method partitioned the keys equally among the parallel processes. Each process in the cluster parallel version tried the same keys as each thread in the SMP parallel version:

| Process | Keys |
|---|---|
| 0 | 0 – 255 |
| 1 | 256 – 511 |
| 2 | 512 – 767 |
| 3 | 768 – 1,023 |

To make a hybrid parallel version, we use both techniques. We use a Range object to divide the key search space among the processes. Within each process, we use a parallel loop to further subdivide the process's key search space among the threads. Specifically, we replace the regular loop of the cluster parallel version with a parallel loop from the lower bound to the upper bound of the process's subrange.

```
Range chunk = new Range(0,maxcounter-1) .subrange(size,rank);
int lb = chunk.lb();
int ub = chunk.ub();
new ParallelTeam().execute (new ParallelRegion()
   {
   public void run() throws Exception
      {
      execute (lb, ub, new IntegerForLoop()
         {
         public void run (int first, int last)
            {
            for (int cntr = first; cntr <= last; ++ cntr)
               {
               // Try key
               }
            }
         });
      }
   });
```

Here are the keys each process and each thread tries in the hybrid parallel version with *Kp* = 4 processes and *Kt* = 4 threads per process, a total of *K* = *Kp*·*Kt* = 16 processors:

| Process | Keys | Thread | Keys |
|---|---|---|---|
| 0 | 0 – 255 | 0 | 0 – 63 |
| | | 1 | 64 – 127 |
| | | 2 | 128 – 191 |
| | | 3 | 192 – 255 |
| 1 | 256 – 511 | 0 | 256 – 319 |
| | | 1 | 320 – 383 |
| | | 2 | 384 – 447 |
| | | 3 | 448 – 511 |
| 2 | 512 – 767 | 0 | 512 – 575 |
| | | 1 | 576 – 639 |
| | | 2 | 640 – 703 |
| | | 3 | 704 – 767 |
| 3 | 768 – 1,023 | 0 | 768 – 831 |
| | | 1 | 832 – 895 |
| | | 2 | 896 – 959 |
| | | 3 | 960 – 1,023 |

Because each loop iteration takes the same amount of time, using the `subrange()` method to partition the keys equally among the processes and using the parallel loop's default fixed schedule to partition each process's subrange equally among the threads results in a balanced load.

## 32.2 Parallel Key Search Program

Here is the code for the hybrid parallel version of the AES key search program, class edu.rit.hyb.keysearch.FindKeyHyb.

```
package edu.rit.hyb.keysearch;
import edu.rit.crypto.blockcipher.AES256CipherSmp;
import edu.rit.pj.Comm;
import edu.rit.pj.IntegerForLoop;
import edu.rit.pj.ParallelRegion;
import edu.rit.pj.ParallelTeam;
import edu.rit.util.Hex;
import edu.rit.util.Range;
public class FindKeyHyb
   {
   // World communicator.
   static Comm world;
   static int size;
   static int rank;

   // Command line arguments.
   static byte[] plaintext;
   static byte[] ciphertext;
   static byte[] partialkey;
   static int n;

   // The least significant 32 bits of the partial key.
   static int keylsbs;

   // The maximum value for the missing key bits counter.
   static int maxcounter;

   // The complete key.
   static byte[] foundkey;

   // Chunk of the search space this process will do.
   static Range chunk;

   /**
    * AES partial key search main program.
    */
   public static void main
      (String[] args)
      throws Exception
      {
      // Start timing.
      long t1 = System.currentTimeMillis();

      // Initialize PJ middleware.
      Comm.init (args);
      world = Comm.world();
      size = world.size();
      rank = world.rank();

      // Parse command line arguments.
      if (args.length != 4) usage();
      plaintext = Hex.toByteArray (args[0]);
      ciphertext = Hex.toByteArray (args[1]);
      partialkey = Hex.toByteArray (args[2]);
      n = Integer.parseInt (args[3]);

      // Make sure n is not too small or too large.
      if (n < 0)
         {
         System.err.println ("n = " + n + " is too small");
         System.exit (1);
         }
      if (n > 30)
         {
         System.err.println ("n = " + n + " is too large");
         System.exit (1);
         }

      // Set up program shared variables for doing trial
      // encryptions.
      keylsbs =
         ((partialkey[28] & 0xFF) << 24) |
         ((partialkey[29] & 0xFF) << 16) |
         ((partialkey[30] & 0xFF) <<  8) |
         ((partialkey[31] & 0xFF)      );
      maxcounter = (1 << n) - 1;

      // Determine which chunk of the search space this process
      // will do.
      chunk = new Range (0, maxcounter) .subrange (size, rank);

      // Do trial encryptions in parallel threads.
```

```
new ParallelTeam().execute (new ParallelRegion()
   {
   public void run() throws Exception
      {
      execute (chunk.lb(), chunk.ub(), new IntegerForLoop()
         {
         // Thread local variables.
         byte[] trialkey;
         byte[] trialciphertext;
         AES256CipherSmp cipher;
         long p0, p1, p2, p3, p4, p5, p6, p7;
         long p8, p9, pa, pb, pc, pd, pe, pf;

         // Set up thread local variables. Extra padding to
         // avert cache interference.
         public void start()
            {
            trialkey = new byte [32+128];
            System.arraycopy
               (partialkey, 0, trialkey, 0, 32);
            trialciphertext = new byte [16+128];
            cipher = new AES256CipherSmp (trialkey);
            }

         // Try every possible combination of low-order key
         // bits.
         public void run (int first, int last)
            {
            for (int counter = first; counter <= last;
                     ++ counter)
               {
               // Fill in low-order key bits.
               int lsbs = keylsbs | counter;
               trialkey[28] = (byte) (lsbs >>> 24);
               trialkey[29] = (byte) (lsbs >>> 16);
               trialkey[30] = (byte) (lsbs >>>  8);
               trialkey[31] = (byte) (lsbs       );

               // Try the key.
               cipher.setKey (trialkey);
               cipher.encrypt (plaintext, trialciphertext);

               // If the result equals the ciphertext, we
               // found the key.
               if (match (ciphertext, trialciphertext))
```

```
                     {
                     foundkey = new byte [32];
                     System.arraycopy
                        (trialkey, 0, foundkey, 0, 32);
                     }
                  }
               }
            });
         }
      });

   // If we found the key, print it.
   if (foundkey != null)
      {
      System.out.println (Hex.toString (foundkey));
      }

   // Stop timing.
   long t2 = System.currentTimeMillis();
   System.out.println ((t2-t1) + " msec " + rank);
   }

/**
 * Returns true if the two byte arrays match.
 */
private static boolean match
   (byte[] a,
    byte[] b)
   {
   boolean matchsofar = true;
   int n = a.length;
   for (int i = 0; i < n; ++ i)
      {
      matchsofar = matchsofar && a[i] == b[i];
      }
   return matchsofar;
   }
}
```

## 32.3 Parallel Program Performance

In Part IV, we measure our hybrid parallel programs' running times on the "tardis" computer, the same 40-processor hybrid parallel computer we used in Part III. Each of this computer's ten backend machines has two 2.6-GHz AMD Opteron 2218 dual-core CPU chips and 8 GB of main memory. The backend machines are connected by a 1-Gbps switched Ethernet. Unlike our cluster parallel programs where we ran as many as four processes on each node, for our hybrid parallel programs, we will run only one process on each node. However, we will run as many as four threads in each process. Specifically, we will measure the running time for every combination of *Kp* processes and *Kt* threads per process, where *Kp* varies from 1 to 10, and *Kt* varies from 1 to 4. Thus, the number of processors *K* varies from 1 to 40. To get a certain number of processes and threads, we include both the `-Dpj.np` and `-Dpj.nt` flags on the `java` command line:

```
$ java -Dpj.np=10 -Dpj.nt=4 . . .
```

Of course, we will measure the running time of the sequential version as well, to calculate the speedups and efficiencies.

Each process in the parallel program reports its own running time. We'll take the program's overall running time to be the largest of the individual processes' running times.

We'll run the hybrid parallel program on two problem sizes, a small problem and a large problem. For each problem size, we'll do seven program runs and record the minimum of the running time measurements.

To plot the running-time data, we'll do something a little different. Hitherto, our plots have shown one dependent variable on the vertical axis—running time, speedup, efficiency, or *EDSF*—and two independent variables—the number of processors *K* on the horizontal axis and the problem size *N* with separate curves. For our hybrid parallel programs, we have *three* independent variables to consider: the number of processes *Kp*; the number of threads per process *Kt*; and the problem size *N*. That's too many independent variables to show on one plot. So, our plots will show metrics versus *Kp* and *Kt*, with separate plots for different problem sizes.

Table 32.1 (at the end of the chapter) lists, and Figure 32.1 plots, the running time data for the FindKeyHyb program for problem sizes of 128M keys (27 missing key bits) and 1G keys (30 missing key bits). The program experiences efficiencies of 90 percent or more all the way out to 40 processors.

Note that the ideal speedup depends on the number of *processors,* which is the number of processes times the number of threads per process. With one thread per process, the ideal speedup goes up to 10 as the number of processes goes up to 10; with two threads per process, the ideal speedup goes up to 20 as the number of processes goes up to 10; and so on. The ideal efficiency is 1, no matter how many threads or processes there are.

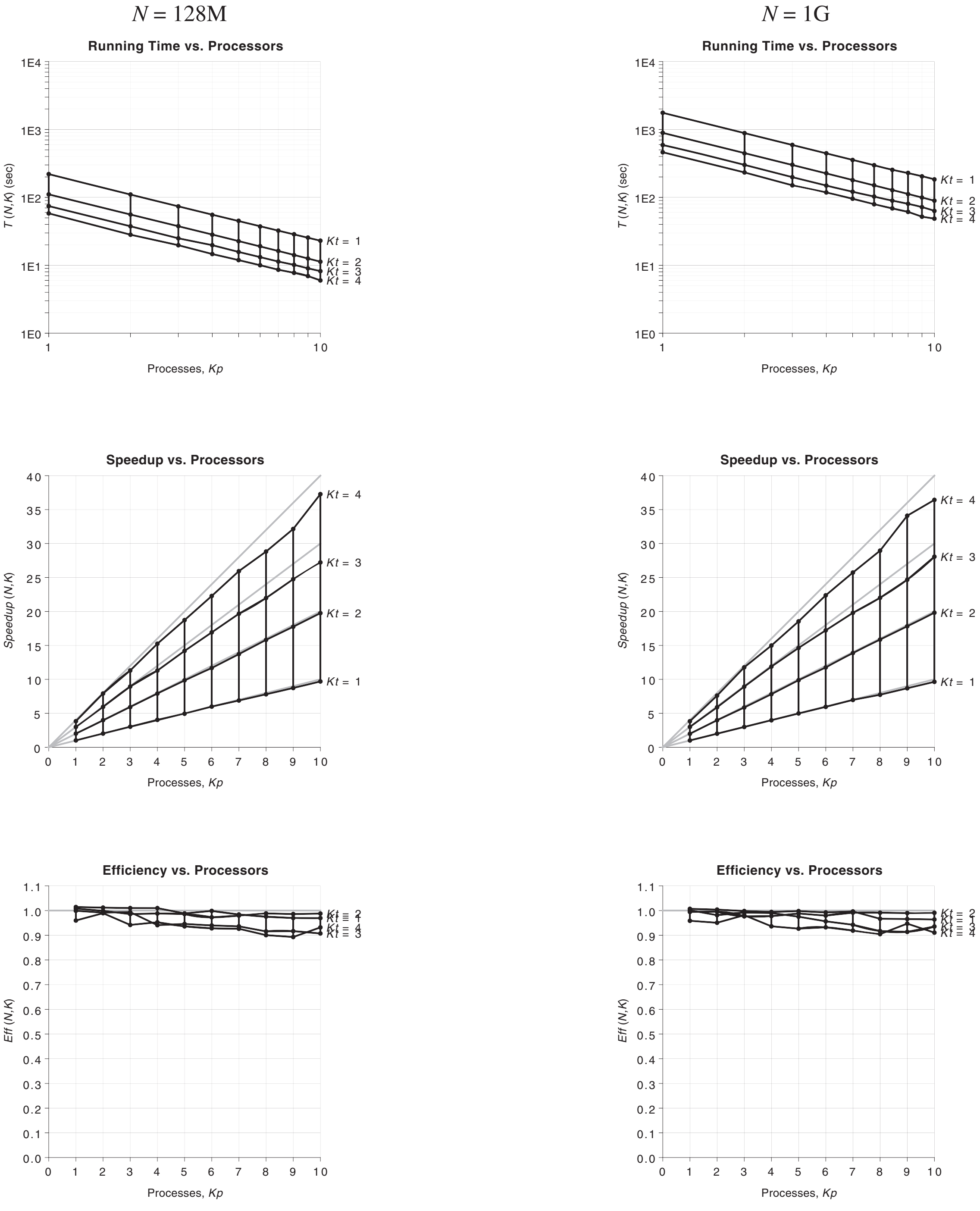
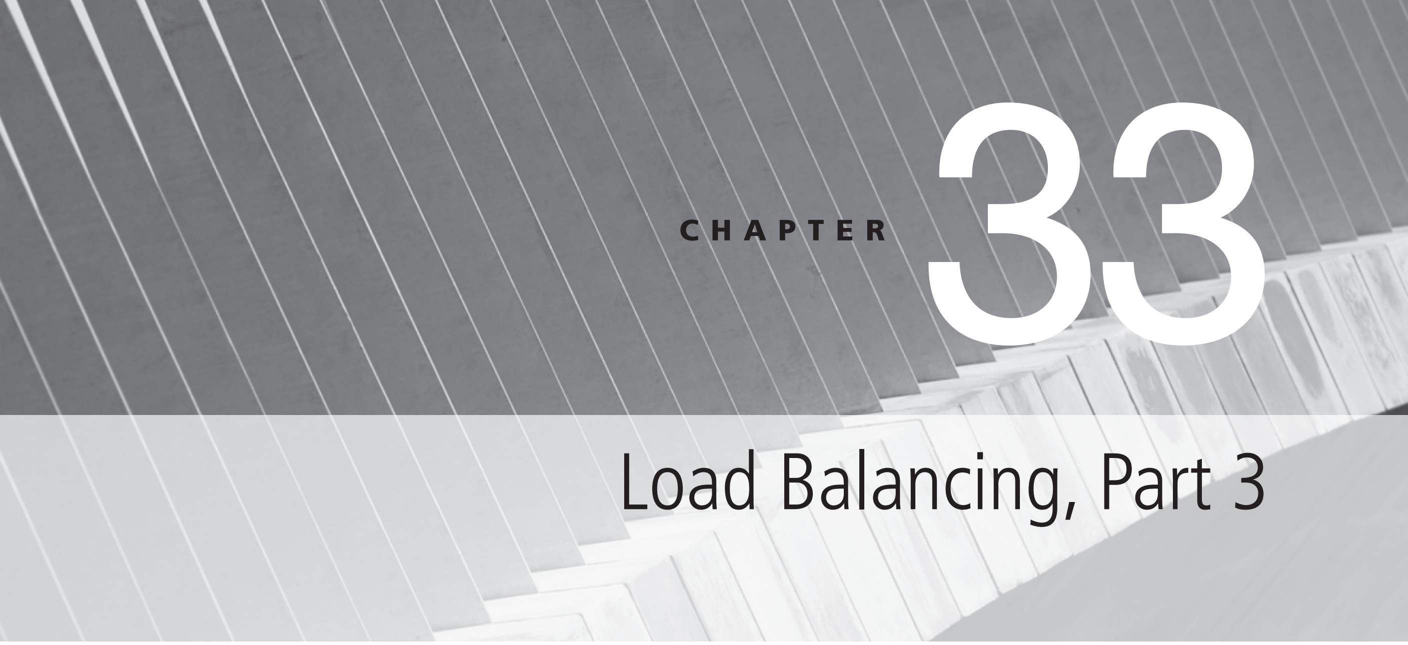


**Figure 32.1** FindKeySeq/FindKeyHyb running-time metrics

**Table 32.1** FindKeySeq/FindKeyHyb running-time metrics

| *N = 128M* | | | | | | *N = 1G* | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| *Kp* | *Kt* | *T* | *Spdup* | *Eff* | *EDSF* | *Kp* | *Kt* | *T* | *Spdup* | *Eff* | *EDSF* |
| seq | seq | 223352 | | | | seq | seq | 1772921 | | | |
| 1 | 1 | 220199 | 1.014 | 1.014 | | 1 | 1 | 1760625 | 1.007 | 1.007 | |
| 1 | 2 | 110768 | 2.016 | 1.008 | 0.006 | 1 | 2 | 891467 | 1.989 | 0.994 | 0.013 |
| 1 | 3 | 74518 | 2.997 | 0.999 | 0.008 | 1 | 3 | 590218 | 3.004 | 1.001 | 0.003 |
| 1 | 4 | 58180 | 3.839 | 0.960 | 0.019 | 1 | 4 | 462374 | 3.834 | 0.959 | 0.017 |
| 2 | 1 | 110364 | 2.024 | 1.012 | 0.002 | 2 | 1 | 883002 | 2.008 | 1.004 | 0.003 |
| 2 | 2 | 55906 | 3.995 | 0.999 | 0.005 | 2 | 2 | 445174 | 3.983 | 0.996 | 0.004 |
| 2 | 3 | 37536 | 5.950 | 0.992 | 0.005 | 2 | 3 | 301088 | 5.888 | 0.981 | 0.005 |
| 2 | 4 | 28219 | 7.915 | 0.989 | 0.004 | 2 | 4 | 233107 | 7.606 | 0.951 | 0.008 |
| 3 | 1 | 73706 | 3.030 | 1.010 | 0.002 | 3 | 1 | 592109 | 2.994 | 0.998 | 0.004 |
| 3 | 2 | 37764 | 5.914 | 0.986 | 0.006 | 3 | 2 | 302549 | 5.860 | 0.977 | 0.006 |
| 3 | 3 | 24973 | 8.944 | 0.994 | 0.003 | 3 | 3 | 198612 | 8.927 | 0.992 | 0.002 |
| 3 | 4 | 19761 | 11.303 | 0.942 | 0.007 | 3 | 4 | 150571 | 11.775 | 0.981 | 0.002 |
| 4 | 1 | 55289 | 4.040 | 1.010 | 0.001 | 4 | 1 | 445779 | 3.977 | 0.994 | 0.004 |
| 4 | 2 | 28255 | 7.905 | 0.988 | 0.004 | 4 | 2 | 226695 | 7.821 | 0.978 | 0.004 |
| 4 | 3 | 19782 | 11.291 | 0.941 | 0.007 | 4 | 3 | 149219 | 11.881 | 0.990 | 0.002 |
| 4 | 4 | 14653 | 15.243 | 0.953 | 0.004 | 4 | 4 | 118349 | 14.980 | 0.936 | 0.005 |
| 5 | 1 | 45193 | 4.942 | 0.988 | 0.007 | 5 | 1 | 355331 | 4.989 | 0.998 | 0.002 |
| 5 | 2 | 22652 | 9.860 | 0.986 | 0.003 | 5 | 2 | 179434 | 9.881 | 0.988 | 0.002 |
| 5 | 3 | 15742 | 14.188 | 0.946 | 0.005 | 5 | 3 | 121203 | 14.628 | 0.975 | 0.002 |
| 5 | 4 | 11929 | 18.723 | 0.936 | 0.004 | 5 | 4 | 95645 | 18.536 | 0.927 | 0.005 |
| 6 | 1 | 37277 | 5.992 | 0.999 | 0.003 | 6 | 1 | 297760 | 5.954 | 0.992 | 0.003 |
| 6 | 2 | 19136 | 11.672 | 0.973 | 0.004 | 6 | 2 | 150823 | 11.755 | 0.980 | 0.003 |
| 6 | 3 | 13204 | 16.915 | 0.940 | 0.005 | 6 | 3 | 102978 | 17.217 | 0.956 | 0.003 |
| 6 | 4 | 10027 | 22.275 | 0.928 | 0.004 | 6 | 4 | 79237 | 22.375 | 0.932 | 0.003 |
| 7 | 1 | 32421 | 6.889 | 0.984 | 0.005 | 7 | 1 | 254176 | 6.975 | 0.996 | 0.002 |
| 7 | 2 | 16286 | 13.714 | 0.980 | 0.003 | 7 | 2 | 127720 | 13.881 | 0.992 | 0.001 |
| 7 | 3 | 11361 | 19.660 | 0.936 | 0.004 | 7 | 3 | 89579 | 19.792 | 0.942 | 0.003 |
| 7 | 4 | 8611 | 25.938 | 0.926 | 0.004 | 7 | 4 | 68908 | 25.729 | 0.919 | 0.004 |
| 8 | 1 | 28620 | 7.804 | 0.976 | 0.006 | 8 | 1 | 229222 | 7.735 | 0.967 | 0.006 |
| 8 | 2 | 14123 | 15.815 | 0.988 | 0.002 | 8 | 2 | 111791 | 15.859 | 0.991 | 0.001 |
| 8 | 3 | 10163 | 21.977 | 0.916 | 0.005 | 8 | 3 | 80546 | 22.011 | 0.917 | 0.004 |
| 8 | 4 | 7751 | 28.816 | 0.900 | 0.004 | 8 | 4 | 61240 | 28.950 | 0.905 | 0.004 |
| 9 | 1 | 25589 | 8.728 | 0.970 | 0.006 | 9 | 1 | 203978 | 8.692 | 0.966 | 0.005 |
| 9 | 2 | 12586 | 17.746 | 0.986 | 0.002 | 9 | 2 | 99504 | 17.818 | 0.990 | 0.001 |
| 9 | 3 | 9024 | 24.751 | 0.917 | 0.004 | 9 | 3 | 71863 | 24.671 | 0.914 | 0.004 |
| 9 | 4 | 6948 | 32.146 | 0.893 | 0.004 | 9 | 4 | 51999 | 34.095 | 0.947 | 0.002 |
| 10 | 1 | 23036 | 9.696 | 0.970 | 0.005 | 10 | 1 | 183908 | 9.640 | 0.964 | 0.005 |
| 10 | 2 | 11306 | 19.755 | 0.988 | 0.001 | 10 | 2 | 89464 | 19.817 | 0.991 | 0.001 |
| 10 | 3 | 8206 | 27.218 | 0.907 | 0.004 | 10 | 3 | 63169 | 28.066 | 0.936 | 0.003 |
| 10 | 4 | 5992 | 37.275 | 0.932 | 0.002 | 10 | 4 | 48670 | 36.427 | 0.911 | 0.003 |

*This page intentionally left blank*



# CHAPTER 33

# Load Balancing, Part 3

in which we learn how to do load balancing in a hybrid parallel program using the master-worker pattern; we see how to combine the master-worker pattern and the parallel loop pattern to do two-level load balancing; and we consider how these alternatives affect the program's performance

# CHAPTER 33 Load Balancing, Part 3

## 33.1 Load Balancing with One-Level Scheduling

Having looked at a hybrid parallel program for a massively parallel problem that required neither message passing nor load balancing, we now turn our attention to a massively parallel problem that does require message passing for load balancing. We'll write a hybrid parallel version of the Mandelbrot Set program from Chapters 11–12 (SMP version) and Chapters 23–24 (cluster version). This program computes the color of each pixel in an image of the Mandelbrot Set and stores the pixel data in an integer matrix. The program then writes the pixel data to a PJG image file.

After several iterations of the design, the cluster parallel Mandelbrot Set program ended up using the *master-worker pattern* for load balancing and the *parallel output files pattern* to reduce the amount of message passing. We will continue to use both patterns in the hybrid parallel program. However, we need to reconsider the manner in which the hybrid parallel program partitions the work among the multiple processes and the multiple threads in each process.

In the cluster parallel program, every process had a single worker thread that ran on the single CPU in each node of the cluster parallel computer (Figure 33.1). Process 0 also had a master thread that shared the CPU with process 0's worker thread. The master used a schedule object to divide the image into chunks (row ranges) to be computed in parallel so as to achieve a balanced load. The master sent messages to the workers with chunks for the workers to compute. The workers sent messages to the master when they finished their chunks. The workers also wrote their chunks of the Mandelbrot Set image to separate output files. However, the workers did not share data with each other or send messages to each other.

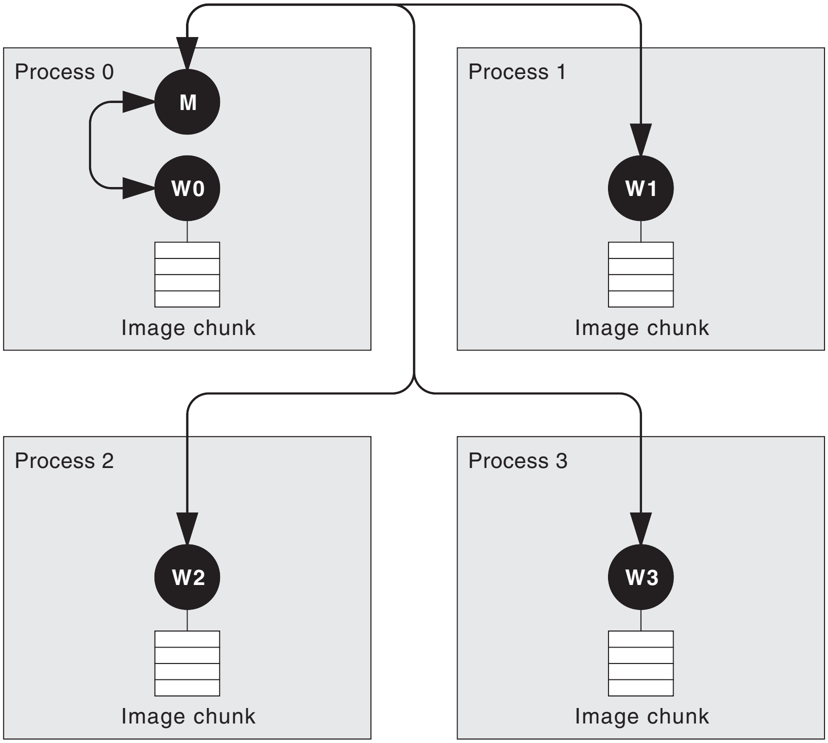


**Figure 33.1** Cluster parallel master-worker program

For a hybrid parallel computer with several CPUs in each node, a straightforward extension of this design is to put several worker threads in each process, one worker for each CPU (Figure 33.2). However, the workers still operate independently of one another, neither sharing data with each other nor sending messages to each other. In addition, process 0 has a master thread that uses a schedule object to partition the image, exactly as in the cluster parallel program. This is the master-worker pattern with **one-level scheduling**, so called because there is just one schedule—the one in the master thread—that determines which chunks each worker thread will compute.

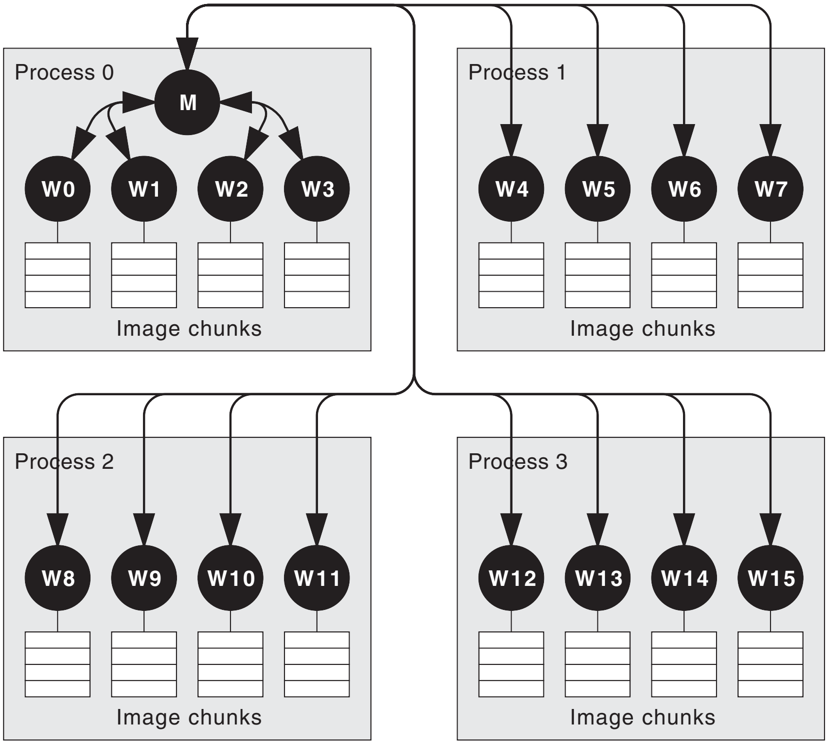


**Figure 33.2** Hybrid parallel master-worker program, one-level scheduling

## 33.2 Hybrid Program with One-Level Scheduling

Here is the source code for class edu.rit.hyb.fractal.MandelbrotSetHyb, the hybrid parallel Mandelbrot Set program with one-level load balancing. It is derived from class edu.rit.clu.fractal.MandelbrotSetClu3, the cluster parallel version.

```
package edu.rit.hyb.fractal;
import edu.rit.color.HSB;
import edu.rit.image.PJGColorImage;
import edu.rit.image.PJGImage;
import edu.rit.io.Files;
import edu.rit.mp.IntegerBuf;
import edu.rit.mp.ObjectBuf;
import edu.rit.mp.buf.ObjectItemBuf;
import edu.rit.pj.Comm;
import edu.rit.pj.CommStatus;
import edu.rit.pj.IntegerSchedule;
import edu.rit.pj.ParallelRegion;
import edu.rit.pj.ParallelTeam;
import edu.rit.pj.PJProperties;
import edu.rit.util.Range;
import java.io.BufferedOutputStream;
import java.io.File;
import java.io.FileOutputStream;
import java.io.IOException;
public class MandelbrotSetHyb
   {
   // Communicator.
   static Comm world;
   static int Kp;
   static int rank;
   static int Kt;
   static int K;

   // Command line arguments.
   static int width;
   static int height;
   static double xcenter;
   static double ycenter;
   static double resolution;
   static int maxiter;
   static double gamma;
   static File filename;

   // Initial pixel offsets from center.
   static int xoffset;
   static int yoffset;

   // Table of hues.
   static int[] huetable;

   /**
    * Mandelbrot Set main program.
    */
   public static void main
      (String[] args)
      throws Exception
      {
      // Start timing.
      long t1 = System.currentTimeMillis();
```

The program needs to know the number of processes *Kp* with which it is running, the number of threads per process *Kt* (which must be the same in all processes), and the number of processors *K* = *Kp*·*Kt*.

```
      // Initialize middleware.
      Comm.init (args);
      world = Comm.world();
      Kp = world.size();
      rank = world.rank();
      Kt = PJProperties.getPjNt();
      if (Kt == 0)
         {
         System.err.println
            ("MandelbrotSetHyb: -Dpj.nt must be specified");
         System.exit (1);
         }
      K = Kp * Kt;

      // Validate command line arguments.
      if (args.length != 8) usage();
      width = Integer.parseInt (args[0]);
      height = Integer.parseInt (args[1]);
      xcenter = Double.parseDouble (args[2]);
      ycenter = Double.parseDouble (args[3]);
      resolution = Double.parseDouble (args[4]);
      maxiter = Integer.parseInt (args[5]);
      gamma = Double.parseDouble (args[6]);
      filename = new File (args[7]);

      // Initial pixel offsets from center.
      xoffset = -(width - 1) / 2;
      yoffset = (height - 1) / 2;

      // Create table of hues for different iteration counts.
      huetable = new int [maxiter+1];
      for (int i = 0; i < maxiter; ++ i)
         {
         huetable[i] = HSB.pack
            (/*hue*/ (float)
               Math.pow (((double)i)/((double)maxiter),gamma),
             /*sat*/ 1.0f,
             /*bri*/ 1.0f);
         }
      huetable[maxiter] = HSB.pack (1.0f, 1.0f, 0.0f);

      long t2 = System.currentTimeMillis();
```

To get multiple threads in each process, we set up a parallel team with *Kt* threads. Each team thread, when it calls the parallel region's `run()` method, executes the `workerSection()` subroutine in parallel, thus acting as a worker thread. Each worker thread has a unique worker index, starting with worker index 0 for the first team thread in the first process and going up to worker index *K*–1 for the last team thread in the last process. The worker index is passed in as the `workerSection()` subroutine's argument. In process 0, the parallel team gets one extra thread. This team thread, when it calls the parallel region's `run()` method, executes the `masterSection()` subroutine, thus acting as the master thread in parallel with the worker threads.

```
// Set up parallel team: Kt+1 threads in process 0, Kt
// threads in processes 1 and up.
ParallelTeam team = new ParallelTeam (rank == 0 ? Kt+1 : Kt);

// Every parallel team thread runs the worker section, except
// thread Kt (which exists only in process 0) runs the master
// section.
team.execute (new ParallelRegion()
   {
   public void run() throws Exception
      {
      if (getThreadIndex() == Kt)
         {
         masterSection();
         }
      else
         {
         workerSection (rank * Kt + getThreadIndex());
         }
      }
   });

long t3 = System.currentTimeMillis();

// Stop timing.
long t4 = System.currentTimeMillis();
System.out.println ((t2-t1) + " msec pre " + rank);
System.out.println ((t3-t2) + " msec calc " + rank);
System.out.println ((t4-t3) + " msec post " + rank);
System.out.println ((t4-t1) + " msec total " + rank);
}
```

Here is the code for the master thread. It is nearly the same as the cluster parallel version, with a few changes.

```
/**
 * Perform the master section.
 */
private static void masterSection()
   throws IOException
   {
   int process, thread, worker;
   Range range;

   // Set up a schedule object to divide the row range into
   // chunks.
   IntegerSchedule schedule = IntegerSchedule.runtime();
   schedule.start (K, new Range (0, height-1));
```

The master needs to send messages to each worker *thread.* However, the communicator only lets us send a message to a *process* at a certain rank. To direct the message to a specific thread within the process, we will specify the worker index as the message *tag.*

```
   // Send initial chunk range to each worker. If range is null,
   // no more work for that worker. Keep count of active workers.
   int activeWorkers = K;
   for (process = 0; process < Kp; ++ process)
      {
      for (thread = 0; thread < Kt; ++ thread)
         {
         worker = process * Kt + thread;
         range = schedule.next (worker);
         world.send (process, worker, ObjectBuf.buffer (range));
         if (range == null) — activeWorkers;
         }
      }
```

The master now waits to receive a message from any process (rank = null, or wildcard) and any worker thread within the process (tag = null, or wildcard). The actual process (rank) and worker index (tag) that sent the message are obtained from the CommStatus object the `receive()` method returns. The master then directs its reply to the same process and worker index.

```
   // Repeat until all workers have finished.
   while (activeWorkers > 0)
      {
      // Receive an empty message from any worker.
      CommStatus status =
         world.receive (null, null, IntegerBuf.emptyBuffer());
      process = status.fromRank;
      worker = status.tag;

      // Send next chunk range to that specific worker. If null,
      // no more work.
      range = schedule.next (worker);
      world.send (process, worker, ObjectBuf.buffer (range));
      if (range == null) — activeWorkers;
      }
   }
```

Here is the code for the worker threads. The argument is the worker index. This code also is nearly the same as the cluster parallel version, with a few changes.

```
/**
 * Perform the worker section.
 */
private static void workerSection
   (int worker)
   throws IOException
   {
```

Each worker gets its own per-thread variables: the pixel matrix row references; the PJG image and writer objects; and the storage for the current pixel matrix row slice. These are now local variables of the `workerSection()` method instead of static global variables.

```
   // Image matrix. Allocate storage for pixel matrix row
   // references only.
   int[][] matrix = new int [height] [];

   // Prepare to write image row slices to per-worker PJG image
   // file.
   PJGColorImage image =
      new PJGColorImage (height, width, matrix);
   PJGImage.Writer writer =
      image.prepareToWrite
         (new BufferedOutputStream
            (new FileOutputStream
               (Files.fileForRank (filename, worker))));

   // Storage for matrix row slice.
   int[][] slice = null;
```

```
      // Process chunks from master.
      for (;;)
         {
         // Receive chunk range from master. If null, no more work.
         ObjectItemBuf<Range> rangeBuf = ObjectBuf.buffer();
```

Here is where we ensure each message from the master ends up in the proper worker. In the `receive()` method call, the worker specifies rank 0 as the source process and the worker's own index as the message tag. The worker will therefore receive only those messages tagged with its own worker index, not messages tagged with the other worker threads' indexes.

```
         world.receive (0, worker, rangeBuf);
         Range range = rangeBuf.item;
         if (range == null) break;
         int lb = range.lb();
         int ub = range.ub();
         int len = range.length();

         // Allocate storage for matrix row slice if necessary.
         if (slice == null || slice.length < len)
            {
            slice = new int [len] [width];
            }

         // Compute all rows and columns in slice.
         for (int r = lb; r <= ub; ++ r)
            {
            int[] slice_r = slice[r-lb];
            double y = ycenter + (yoffset - r) / resolution;

            for (int c = 0; c < width; ++ c)
               {
               double x = xcenter + (xoffset + c) / resolution;

               // Iterate until convergence.
               int i = 0;
               double aold = 0.0;
               double bold = 0.0;
               double a = 0.0;
               double b = 0.0;
               double zmagsqr = 0.0;
               while (i < maxiter && zmagsqr <= 4.0)
                  {
                  ++ i;
```

```
                     a = aold*aold - bold*bold + x;
                     b = 2.0*aold*bold + y;
                     zmagsqr = a*a + b*b;
                     aold = a;
                     bold = b;
                     }

                  // Record number of iterations for pixel.
                  slice_r[c] = huetable[i];
                  }
               }
```

And, when the worker replies to the master, it tags the message with its own worker index to let the master know which worker replied.

```
            // Report completion of slice to master.
            world.send (0, worker, IntegerBuf.emptyBuffer());

            // Set full pixel matrix rows to refer to slice rows.
            System.arraycopy (slice, 0, matrix, lb, len);

            // Write row slice of full pixel matrix to image file.
            writer.writeRowSlice (range);
            }

         // Close image file.
         writer.close();
         }
      }
```

## 33.3 Program Performance with One-Level Scheduling

Table 33.1 (at the end of the chapter) lists, and Figure 33.3 plots, the MandelbrotSetHyb program's running-time metrics on the "tardis" hybrid parallel computer for two problem sizes: an 8,960 × 8,960-pixel image ($N$ = 80M pixels) and a 25,600 × 25,600-pixel image ($N$ = 160M pixels). To balance the load, the master used a dynamic schedule with a chunk size of 10. The commands for the two problem sizes were

```
$ java -Dpj.np=$KP -Dpj.nt=$KT -Dpj.schedule="dynamic(10)" \
  edu.rit.hyb.fractal.MandelbrotSetHyb 8960 8960 -0.75 0.0 3360 \
  1000 0.4 image.pjg
$ java -Dpj.np=$KP -Dpj.nt=$KT -Dpj.schedule="dynamic(10)" \
  edu.rit.hyb.fractal.MandelbrotSetHyb 25600 25600 -0.75 0.0 9600 \
  1000 0.4 image.pjg
```

where `$KP` and `$KT` are the number of processes and the number of threads per process.

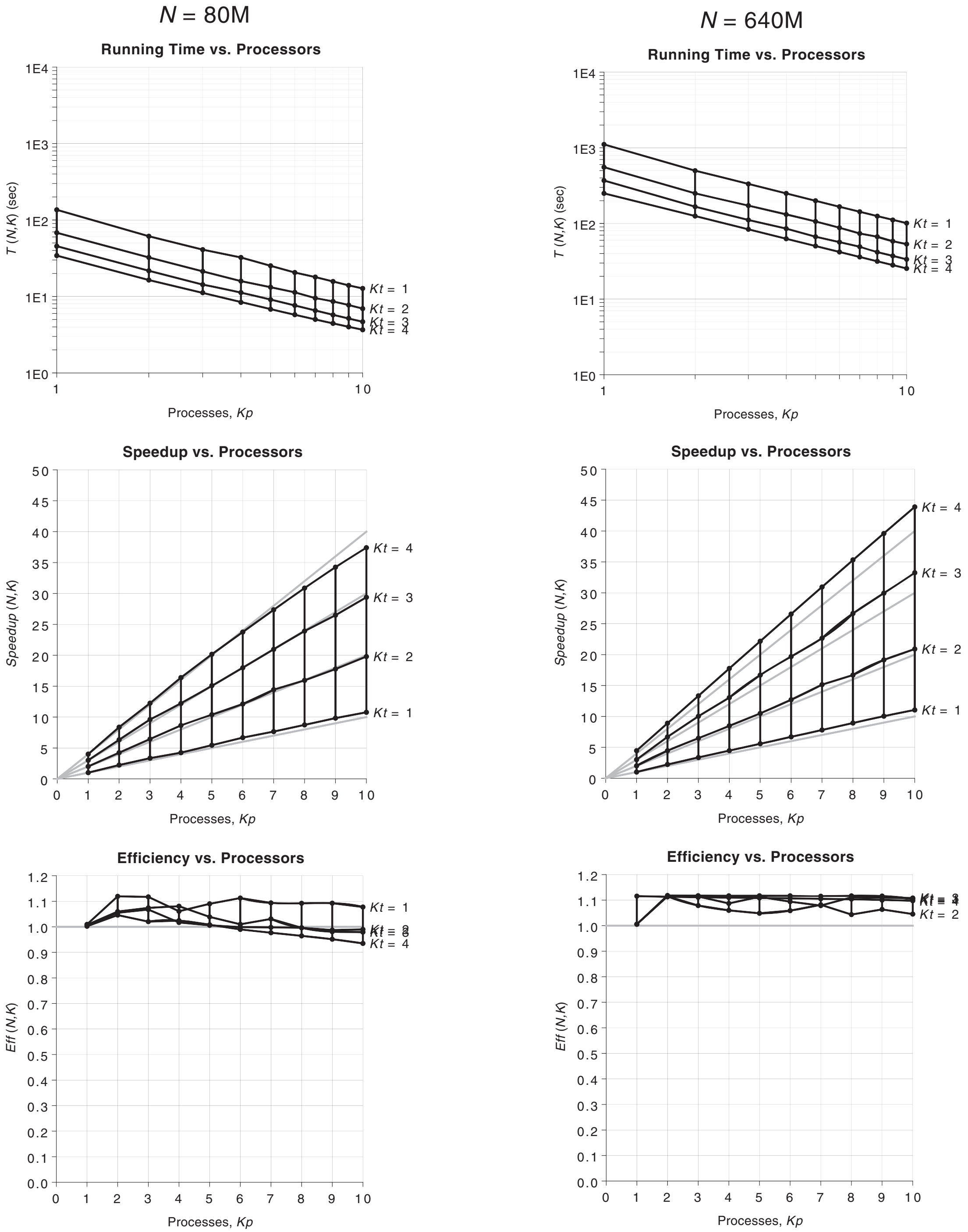


**Figure 33.3** MandelbrotSetSeq/MandelbrotSetHyb running-time metrics

Comparing the hybrid parallel MandelbrotSetHyb program's running times to those of the cluster parallel MandelbrotSetClu3 program in Chapter 24, we see that the hybrid program's running times are a few seconds shorter—enough shorter to push the speedups over the line into superlinear territory for the larger problem size. The reduced running times are due to the JIT compiler effect. With as many as four threads executing the code in each process, the JVM can optimize the machine code more quickly than in the cluster version, which has only one thread executing in each process.

## 33.4 Load Balancing with Two-Level Scheduling

Another way to utilize the multiple CPUs in each node of a hybrid parallel computer is to go back to one worker in each process, like the cluster parallel version. The master still divides the image into chunks (row slices) according to some schedule and sends the chunks to the workers. But then each worker uses a *parallel team* to calculate the rows of the slice in parallel in multiple threads (Figure 33.4). The parallel team uses its own schedule (not necessarily the same as the master's schedule) to divide the rows of the slice among the team threads. This is the master-worker pattern with **two-level scheduling**—the master schedule that determines which chunks each worker will compute and the parallel team schedule that determines which rows of the chunk each team thread will compute.

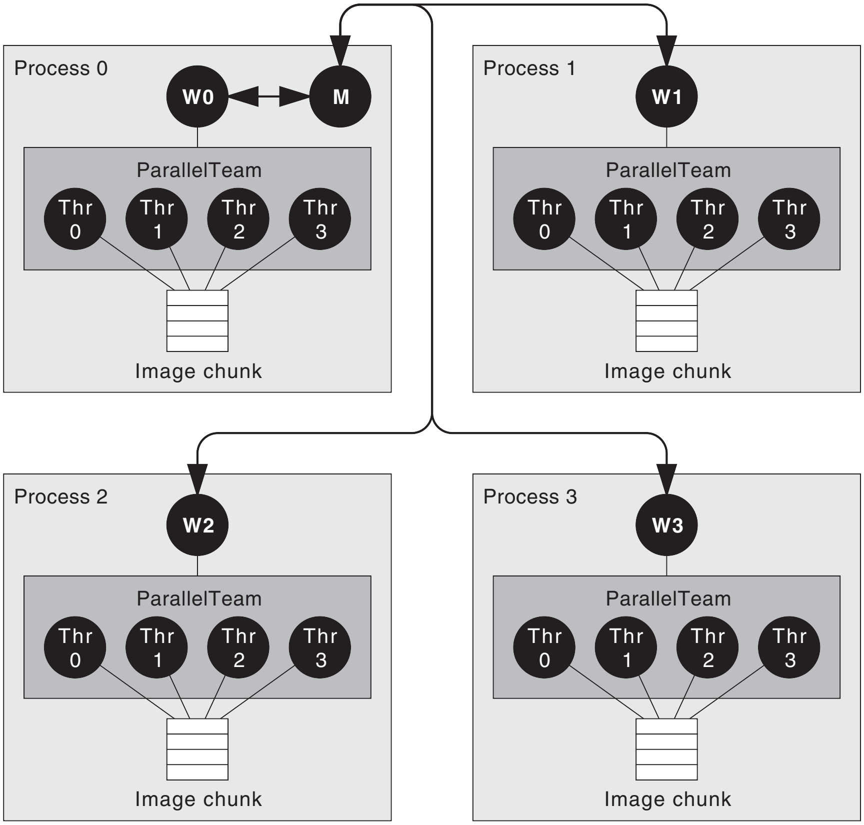


**Figure 33.4** Hybrid parallel master-worker program, two-level scheduling

Why contemplate a two-level scheduling scheme, when the one-level scheduling scheme already gives great performance? With one-level scheduling, we had to make the chunks rather small (10 rows) to ensure there would be enough short-running-time chunks to balance the long-running-time chunks. For each chunk, the master had to send a message and the worker had to send back a reply. Consequently, the program had to send 1,792 messages to compute the 8,960-row image, and the program had to send 5,120 messages to compute the 25,600-row image. On the "tardis" computer, the message latency alone consumed 0.4 seconds and 1.1 seconds, respectively—or, putting it another way, 10 percent and 4 percent of the running time on 40 processors. That's a fairly hefty message-passing overhead for a massively parallel problem, which doesn't need to send messages at all except to do load balancing. We want to cut down that overhead, if possible.

With two-level scheduling, we don't have to be so concerned about making the chunks small, because the parallel team does a second level of load balancing within each chunk. Indeed, the chunks have to be somewhat larger to ensure there are enough short-running-time rows to balance the long-running-time rows in each chunk. If we use, say, 100-row chunks instead of 10-row chunks, we can cut down the message-passing overhead by a factor of 10. On the other hand, with larger chunks, the workers will spend more time writing each chunk to the output file, file I/O might become the bottleneck instead of message passing, and the performance might show no improvement, or might even get worse. The only way to know whether one-level or two-level scheduling performs better is to implement and measure both alternatives.

## 33.5 Hybrid Program with Two-Level Scheduling

Here is the source code for class edu.rit.hyb.fractal.MandelbrotSetHyb2, the hybrid parallel Mandelbrot Set program with two-level load balancing. The master schedule is specified by the `-Dpj.schedule` flag. The parallel team schedule is specified by the last command-line argument; if omitted, the parallel team uses a fixed schedule.

```
package edu.rit.hyb.fractal;
import edu.rit.color.HSB;
import edu.rit.image.PJGColorImage;
import edu.rit.image.PJGImage;
import edu.rit.io.Files;
import edu.rit.mp.IntegerBuf;
import edu.rit.mp.ObjectBuf;
import edu.rit.mp.buf.ObjectItemBuf;
import edu.rit.pj.Comm;
import edu.rit.pj.CommStatus;
import edu.rit.pj.IntegerForLoop;
import edu.rit.pj.IntegerSchedule;
import edu.rit.pj.ParallelRegion;
import edu.rit.pj.ParallelSection;
import edu.rit.pj.ParallelTeam;
import edu.rit.util.Range;
import java.io.BufferedOutputStream;
import java.io.File;
import java.io.FileOutputStream;
import java.io.IOException;
public class MandelbrotSetHyb2
   {
   // Communicator.
   static Comm world;
   static int size;
   static int rank;

   // Command line arguments.
   static int width;
   static int height;
   static double xcenter;
   static double ycenter;
   static double resolution;
   static int maxiter;
   static double gamma;
   static File filename;
   static IntegerSchedule thrschedule;

   // Initial pixel offsets from center.
   static int xoffset;
   static int yoffset;

   // Image matrix.
   static int[][] matrix;
   static PJGColorImage image;
   static PJGImage.Writer writer;

   // Storage for matrix row slice.
   static int[][] slice;

   // Table of hues.
   static int[] huetable;

   /**
    * Mandelbrot Set main program.
    */
   public static void main
      (String[] args)
      throws Exception
      {
      // Start timing.
      long t1 = System.currentTimeMillis();
      // Initialize middleware.
      Comm.init (args);
      world = Comm.world();
      size = world.size();
      rank = world.rank();

      // Validate command line arguments.
      if (args.length < 8 || args.length > 9) usage();
      width = Integer.parseInt (args[0]);
      height = Integer.parseInt (args[1]);
      xcenter = Double.parseDouble (args[2]);
      ycenter = Double.parseDouble (args[3]);
      resolution = Double.parseDouble (args[4]);
      maxiter = Integer.parseInt (args[5]);
      gamma = Double.parseDouble (args[6]);
      filename = new File (args[7]);
      thrschedule =
         args.length == 9 ?
            IntegerSchedule.parse (args[8]) :
            IntegerSchedule.fixed();

      // Initial pixel offsets from center.
      xoffset = -(width - 1) / 2;
      yoffset = (height - 1) / 2;

      // Allocate storage for pixel matrix row references only.
      matrix = new int [height] [];

      // Prepare to write image row slices to per-worker PJG image
      // file.
      image = new PJGColorImage (height, width, matrix);
      writer =
         image.prepareToWrite
            (new BufferedOutputStream
               (new FileOutputStream
                  (Files.fileForRank (filename, rank))));

      // Create table of hues for different iteration counts.
      huetable = new int [maxiter+1];
      for (int i = 0; i < maxiter; ++ i)
         {
         huetable[i] = HSB.pack
            (/*hue*/ (float)
               Math.pow (((double)i)/((double)maxiter),gamma),
             /*sat*/ 1.0f,
             /*bri*/ 1.0f);
         }
      huetable[maxiter] = HSB.pack (1.0f, 1.0f, 0.0f);

      long t2 = System.currentTimeMillis();
```

We're back to the design of the cluster parallel version, with a parallel team of two threads executing the master section and the worker section (which are located in subroutines). Only process 0, the master process, sets up this team. The other processes just execute the worker section.

```
   // In master process, run master section and worker section
   // in parallel.
   if (rank == 0)
      {
      new ParallelTeam(2).execute (new ParallelRegion()
         {
         public void run() throws Exception
            {
            execute (new ParallelSection()
               {
               public void run() throws Exception
                  {
                  masterSection();
                  }
               },
            new ParallelSection()
               {
               public void run() throws Exception
                  {
                  workerSection();
                  }
               });
            }
         });
      }

   // In worker process, run only worker section.
   else
      {
      workerSection();
      }

   long t3 = System.currentTimeMillis();
   // Close image file.
   writer.close();

   // Stop timing.
   long t4 = System.currentTimeMillis();
   System.out.println ((t2-t1) + " msec pre " + rank);
   System.out.println ((t3-t2) + " msec calc " + rank);
   System.out.println ((t4-t3) + " msec post " + rank);
   System.out.println ((t4-t1) + " msec total " + rank);
   }
```

The master section is identical to that of the cluster parallel version. Unlike the first hybrid parallel version, we don’t need to use message tags to direct messages to specific worker threads, because there is only one worker in each process.

```
/**
 * Perform the master section.
 */
private static void masterSection()
   throws IOException
   {
   int worker;
   Range range;

   // Set up a schedule object to divide the row range into
   // chunks.
   IntegerSchedule schedule = IntegerSchedule.runtime();
   schedule.start (size, new Range (0, height-1));

   // Send initial chunk range to each worker. If range is null,
   // no more work for that worker. Keep count of active workers.
   int activeWorkers = size;
   for (worker = 0; worker < size; ++ worker)
      {
      range = schedule.next (worker);
      world.send (worker, ObjectBuf.buffer (range));
      if (range == null) — activeWorkers;
      }

   // Repeat until all workers have finished.
   while (activeWorkers > 0)
      {
      // Receive an empty message from any worker.
      CommStatus status =
         world.receive (null, IntegerBuf.emptyBuffer());
      worker = status.fromRank;

      // Send next chunk range to that specific worker. If null,
      // no more work.
      range = schedule.next (worker);
      world.send (worker, ObjectBuf.buffer (range));
      if (range == null) — activeWorkers;
      }
   }

/**
 * Perform the worker section.
 */
private static void workerSection()
   throws Exception
   {
```

The worker begins by setting up the parallel team that will be used later to calculate the rows of each chunk.

```
   // Parallel team to calculate each slice in multiple threads.
   ParallelTeam team = new ParallelTeam();
```

At this point, the one worker thread is running. It receives a chunk from the master and allocates storage to hold the row slice.

```
   // Process chunks from master.
   for (;;)
      {
      // Receive chunk range from master. If null, no more work.
      ObjectItemBuf<Range> rangeBuf = ObjectBuf.buffer();
      world.receive (0, rangeBuf);
      Range range = rangeBuf.item;
      if (range == null) break;
      final int lb = range.lb();
      final int ub = range.ub();
      final int len = range.length();

      // Allocate storage for matrix row slice if necessary.
      if (slice == null || slice.length < len)
         {
         slice = new int [len] [width];
         }
```

Now the parallel team takes over to calculate the rows of the chunk.

```
// Compute rows of slice in parallel threads.
team.execute (new ParallelRegion()
   {
   public void run() throws Exception
      {
      execute (lb, ub, new IntegerForLoop()
         {
```

The parallel team uses the schedule specified as the last command-line argument.

```
         // Use the thread-level loop schedule.
         public IntegerSchedule schedule()
            {
            return thrschedule;
            }

         // Compute all rows and columns in slice.
         public void run (int first, int last)
            {
            for (int r = first; r <= last; ++ r)
               {
               int[] slice_r = slice[r-lb];
               double y =
                  ycenter + (yoffset - r) / resolution;

               for (int c = 0; c < width; ++ c)
                  {
                  double x =
                     xcenter + (xoffset + c) / resolution;

                  // Iterate until convergence.
                  int i = 0;
                  double aold = 0.0;
                  double bold = 0.0;
                  double a = 0.0;
                  double b = 0.0;
                  double zmagsqr = 0.0;
                  while (i < maxiter && zmagsqr <= 4.0)
                     {
                     ++ i;
                     a = aold*aold - bold*bold + x;
                     b = 2.0*aold*bold + y;
```

```
                           zmagsqr = a*a + b*b;
                           aold = a;
                           bold = b;
                           }

                        // Record number of iterations for
                        // pixel.
                        slice_r[c] = huetable[i];
                        }
                     }
                  }
               });
            }
         });
```

Here we are back in the single worker thread again.

```
      // Report completion of slice to master.
      world.send (0, IntegerBuf.emptyBuffer());

      // Set full pixel matrix rows to refer to slice rows.
      System.arraycopy (slice, 0, matrix, lb, len);

      // Write row slice of full pixel matrix to image file.
      writer.writeRowSlice (range);
      }
   }
}
```

## 33.6 Program Performance with Two-Level Scheduling

Table 33.2 (at the end of the chapter) lists, and Figure 33.5 plots, the MandelbrotSetHyb2 program's running-time metrics on the "tardis" hybrid parallel computer for two problem sizes, an 8,960 × 8,960-pixel image (N = 80M pixels) and a 25,600 × 25,600-pixel image (N = 160M pixels). To balance the load, the master used a dynamic schedule with a chunk size of 100, and the parallel teams used a dynamic schedule with a chunk size of 1. The commands for the two problem sizes were

```
$ java -Dpj.np=$KP -Dpj.nt=$KT -Dpj.schedule="dynamic(100)" \
  edu.rit.hyb.fractal.MandelbrotSetHyb2 8960 8960 -0.75 0.0 3360 \
  1000 0.4 image.pjg "dynamic(1)"
$ java -Dpj.np=$KP -Dpj.nt=$KT -Dpj.schedule="dynamic(100)" \
  edu.rit.hyb.fractal.MandelbrotSetHyb2 25600 25600 -0.75 0.0 9600 \
  1000 0.4 image.pjg "dynamic(1)"
```

where `$KP` and `$KT` are the number of processes and the number of threads per process.

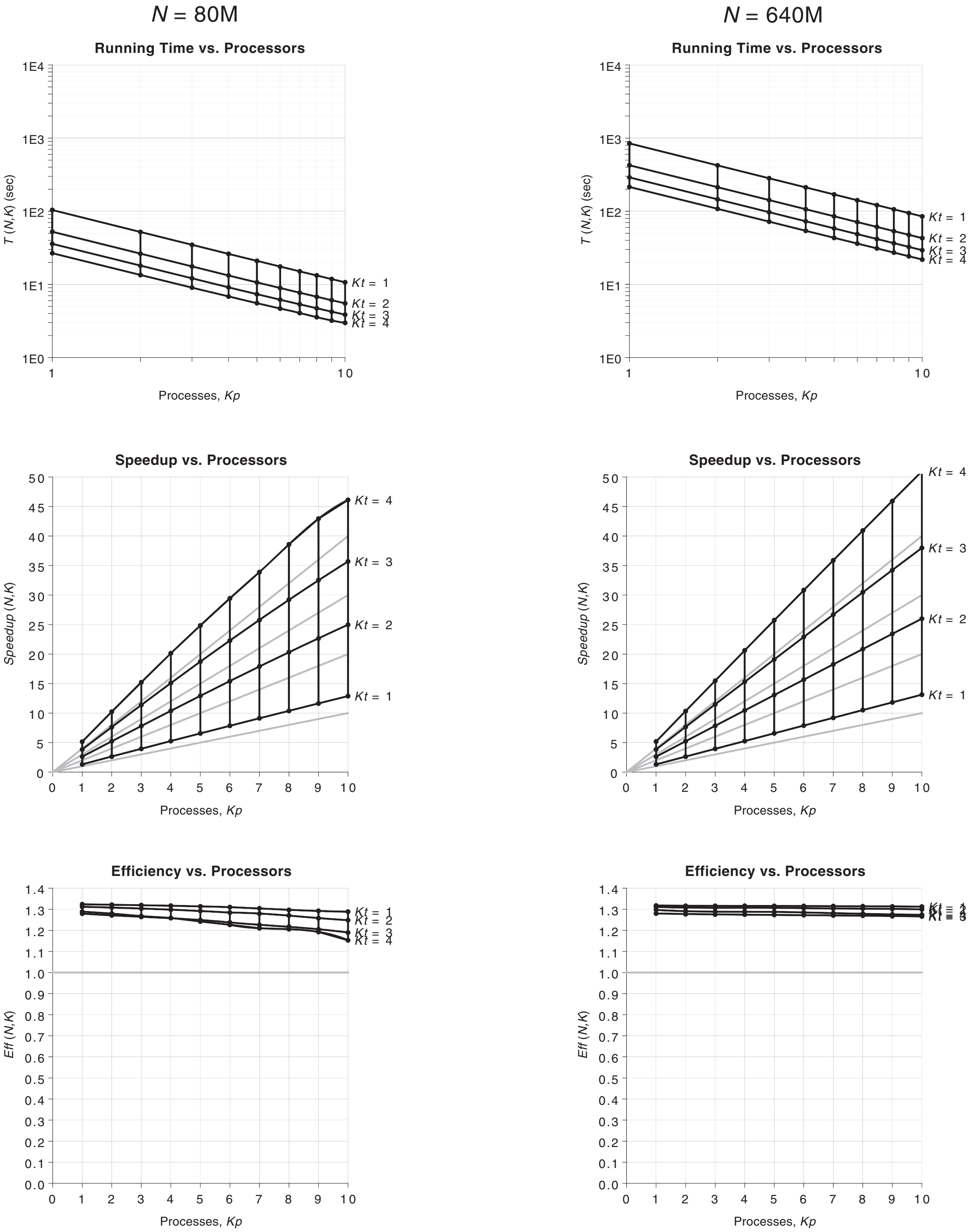


**Figure 33.5** MandelbrotSetSeq/MandelbrotSetHyb2 running-time metrics

Compared to the hybrid parallel version with one-level scheduling, the version with two-level scheduling is about 15–20 percent faster on 40 processors. This confirms our hypothesis that the two-level scheduling would improve the program's performance.

Compared to the cluster parallel version in Chapter 24, the hybrid parallel version with two-level scheduling is faster still—about 25 percent faster on 40 processors. Running a plain cluster program on a hybrid parallel computer, with multiple processes on each node, does not take full advantage of the hybrid parallel computer's capabilities. To get the best performance on a hybrid parallel computer, a parallel program should run with one process per node and multiple threads per process, using message passing parallel programming techniques between the processes and shared memory multithreaded parallel programming techniques within each process.

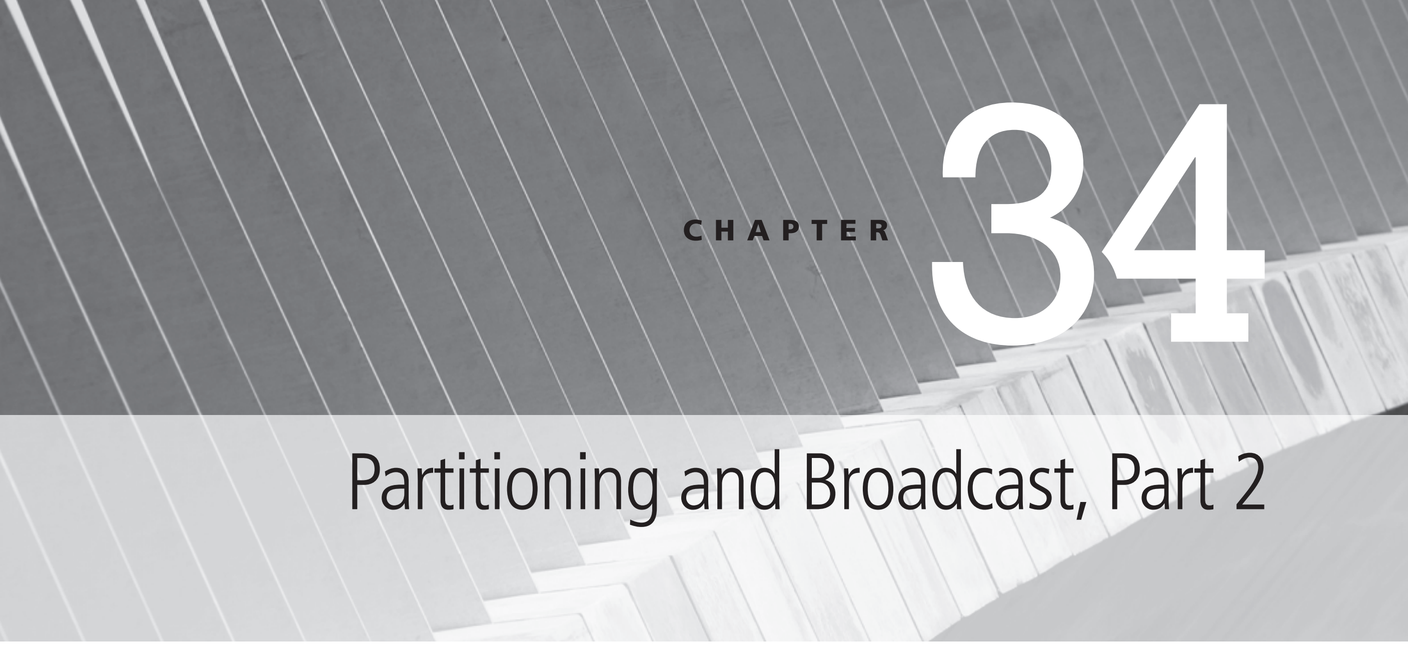
**Table 33.1** MandelbrotSetSeq/MandelbrotSetHyb running-time metrics

| *N* = 80M | | | | | | *N* = 640M | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| *Kp* | *Kt* | *T* | *Spdup* | *Eff* | *EDSF* | *Kp* | *Kt* | *T* | *Spdup* | *Eff* | *EDSF* |
| seq | seq | 137268 | | | | seq | seq | 1131678 | | | |
| 1 | 1 | 135965 | 1.010 | 1.010 | | 1 | 1 | 1109374 | 1.020 | 1.020 | |
| 1 | 2 | 68124 | 2.015 | 1.007 | 0.002 | 1 | 2 | 554599 | 2.041 | 1.020 | 0.000 |
| 1 | 3 | 45494 | 3.017 | 1.006 | 0.002 | 1 | 3 | 369562 | 3.062 | 1.021 | 0.000 |
| 1 | 4 | 34238 | 4.009 | 1.002 | 0.002 | 1 | 4 | 249960 | 4.527 | 1.132 | -0.033 |
| 2 | 1 | 61316 | 2.239 | 1.119 | -0.098 | 2 | 1 | 498782 | 2.269 | 1.134 | -0.101 |
| 2 | 2 | 32384 | 4.239 | 1.060 | -0.016 | 2 | 2 | 250129 | 4.524 | 1.131 | -0.033 |
| 2 | 3 | 21684 | 6.330 | 1.055 | -0.009 | 2 | 3 | 166686 | 6.789 | 1.132 | -0.020 |
| 2 | 4 | 16404 | 8.368 | 1.046 | -0.005 | 2 | 4 | 125299 | 9.032 | 1.129 | -0.014 |
| 3 | 1 | 40956 | 3.352 | 1.117 | -0.048 | 3 | 1 | 332758 | 3.401 | 1.134 | -0.050 |
| 3 | 2 | 21299 | 6.445 | 1.074 | -0.012 | 3 | 2 | 172324 | 6.567 | 1.095 | -0.014 |
| 3 | 3 | 14286 | 9.609 | 1.068 | -0.007 | 3 | 3 | 111230 | 10.174 | 1.130 | -0.012 |
| 3 | 4 | 11211 | 12.244 | 1.020 | -0.001 | 3 | 4 | 83666 | 13.526 | 1.127 | -0.009 |
| 4 | 1 | 32347 | 4.244 | 1.061 | -0.016 | 4 | 1 | 249672 | 4.533 | 1.133 | -0.033 |
| 4 | 2 | 15877 | 8.646 | 1.081 | -0.009 | 4 | 2 | 131538 | 8.603 | 1.075 | -0.007 |
| 4 | 3 | 11247 | 12.205 | 1.017 | -0.001 | 4 | 3 | 85508 | 13.235 | 1.103 | -0.007 |
| 4 | 4 | 8376 | 16.388 | 1.024 | -0.001 | 4 | 4 | 62795 | 18.022 | 1.126 | -0.006 |
| 5 | 1 | 25186 | 5.450 | 1.090 | -0.018 | 5 | 1 | 199704 | 5.667 | 1.133 | -0.025 |
| 5 | 2 | 13213 | 10.389 | 1.039 | -0.003 | 5 | 2 | 106386 | 10.637 | 1.064 | -0.005 |
| 5 | 3 | 9096 | 15.091 | 1.006 | 0.000 | 5 | 3 | 66823 | 16.935 | 1.129 | -0.007 |
| 5 | 4 | 6808 | 20.163 | 1.008 | 0.000 | 5 | 4 | 50309 | 22.495 | 1.125 | -0.005 |
| 6 | 1 | 20562 | 6.676 | 1.113 | -0.019 | 6 | 1 | 166446 | 6.799 | 1.133 | -0.020 |
| 6 | 2 | 11327 | 12.119 | 1.010 | 0.000 | 6 | 2 | 87793 | 12.890 | 1.074 | -0.005 |
| 6 | 3 | 7636 | 17.976 | 0.999 | 0.001 | 6 | 3 | 56647 | 19.978 | 1.110 | -0.005 |
| 6 | 4 | 5778 | 23.757 | 0.990 | 0.001 | 6 | 4 | 41988 | 26.952 | 1.123 | -0.004 |
| 7 | 1 | 17931 | 7.655 | 1.094 | -0.013 | 7 | 1 | 142871 | 7.921 | 1.132 | -0.016 |
| 7 | 2 | 9511 | 14.433 | 1.031 | -0.002 | 7 | 2 | 73651 | 15.365 | 1.098 | -0.005 |
| 7 | 3 | 6550 | 20.957 | 0.998 | 0.001 | 7 | 3 | 49242 | 22.982 | 1.094 | -0.003 |
| 7 | 4 | 5018 | 27.355 | 0.977 | 0.001 | 7 | 4 | 36052 | 31.390 | 1.121 | -0.003 |
| 8 | 1 | 15711 | 8.737 | 1.092 | -0.011 | 8 | 1 | 124862 | 9.063 | 1.133 | -0.014 |
| 8 | 2 | 8612 | 15.939 | 0.996 | 0.001 | 8 | 2 | 66814 | 16.938 | 1.059 | -0.002 |
| 8 | 3 | 5740 | 23.914 | 0.996 | 0.001 | 8 | 3 | 41838 | 27.049 | 1.127 | -0.004 |
| 8 | 4 | 4447 | 30.868 | 0.965 | 0.002 | 8 | 4 | 31574 | 35.842 | 1.120 | -0.003 |
| 9 | 1 | 13960 | 9.833 | 1.093 | -0.009 | 9 | 1 | 111064 | 10.189 | 1.132 | -0.012 |
| 9 | 2 | 7725 | 17.769 | 0.987 | 0.001 | 9 | 2 | 58263 | 19.424 | 1.079 | -0.003 |
| 9 | 3 | 5183 | 26.484 | 0.981 | 0.001 | 9 | 3 | 37248 | 30.382 | 1.125 | -0.004 |
| 9 | 4 | 4007 | 34.257 | 0.952 | 0.002 | 9 | 4 | 28167 | 40.177 | 1.116 | -0.002 |
| 10 | 1 | 12739 | 10.775 | 1.078 | -0.007 | 10 | 1 | 101003 | 11.204 | 1.120 | -0.010 |
| 10 | 2 | 6934 | 19.796 | 0.990 | 0.001 | 10 | 2 | 53353 | 21.211 | 1.061 | -0.002 |
| 10 | 3 | 4672 | 29.381 | 0.979 | 0.001 | 10 | 3 | 33550 | 33.731 | 1.124 | -0.003 |
| 10 | 4 | 3670 | 37.403 | 0.935 | 0.002 | 10 | 4 | 25408 | 44.540 | 1.114 | -0.002 |

**Table 33.2** MandelbrotSetSeq/MandelbrotSetHyb2 running-time metrics

| *N*=80M | | | | | | *N*=640M | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| *Kp* | *Kt* | *T* | *Spdup* | *Eff* | *EDSF* | *Kp* | *Kt* | *T* | *Spdup* | *Eff* | *EDSF* |
| seq | seq | 137614 | | | | seq | seq | 1114794 | | | |
| 1 | 1 | 103960 | 1.324 | 1.324 | | 1 | 1 | 846007 | 1.318 | 1.318 | |
| 1 | 2 | 52450 | 2.624 | 1.312 | 0.009 | 1 | 2 | 425098 | 2.622 | 1.311 | 0.005 |
| 1 | 3 | 35847 | 3.839 | 1.280 | 0.017 | 1 | 3 | 290275 | 3.840 | 1.280 | 0.015 |
| 1 | 4 | 26691 | 5.156 | 1.289 | 0.009 | 1 | 4 | 214850 | 5.189 | 1.297 | 0.005 |
| 2 | 1 | 52078 | 2.642 | 1.321 | 0.002 | 2 | 1 | 423494 | 2.632 | 1.316 | 0.001 |
| 2 | 2 | 26290 | 5.234 | 1.309 | 0.004 | 2 | 2 | 213088 | 5.232 | 1.308 | 0.002 |
| 2 | 3 | 18040 | 7.628 | 1.271 | 0.008 | 2 | 3 | 145526 | 7.660 | 1.277 | 0.006 |
| 2 | 4 | 13442 | 10.238 | 1.280 | 0.005 | 2 | 4 | 107950 | 10.327 | 1.291 | 0.003 |
| 3 | 1 | 34759 | 3.959 | 1.320 | 0.002 | 3 | 1 | 282357 | 3.948 | 1.316 | 0.001 |
| 3 | 2 | 17596 | 7.821 | 1.303 | 0.003 | 3 | 2 | 142136 | 7.843 | 1.307 | 0.002 |
| 3 | 3 | 12095 | 11.378 | 1.264 | 0.006 | 3 | 3 | 97123 | 11.478 | 1.275 | 0.004 |
| 3 | 4 | 9047 | 15.211 | 1.268 | 0.004 | 3 | 4 | 72086 | 15.465 | 1.289 | 0.002 |
| 4 | 1 | 26123 | 5.268 | 1.317 | 0.002 | 4 | 1 | 211714 | 5.266 | 1.316 | 0.000 |
| 4 | 2 | 13250 | 10.386 | 1.298 | 0.003 | 4 | 2 | 106578 | 10.460 | 1.307 | 0.001 |
| 4 | 3 | 9113 | 15.101 | 1.258 | 0.005 | 4 | 3 | 72852 | 15.302 | 1.275 | 0.003 |
| 4 | 4 | 6831 | 20.146 | 1.259 | 0.003 | 4 | 4 | 54066 | 20.619 | 1.289 | 0.002 |
| 5 | 1 | 20952 | 6.568 | 1.314 | 0.002 | 5 | 1 | 169448 | 6.579 | 1.316 | 0.000 |
| 5 | 2 | 10650 | 12.922 | 1.292 | 0.003 | 5 | 2 | 85283 | 13.072 | 1.307 | 0.001 |
| 5 | 3 | 7346 | 18.733 | 1.249 | 0.004 | 5 | 3 | 58352 | 19.105 | 1.274 | 0.002 |
| 5 | 4 | 5539 | 24.845 | 1.242 | 0.003 | 5 | 4 | 43299 | 25.746 | 1.287 | 0.001 |
| 6 | 1 | 17505 | 7.861 | 1.310 | 0.002 | 6 | 1 | 141240 | 7.893 | 1.315 | 0.000 |
| 6 | 2 | 8928 | 15.414 | 1.284 | 0.003 | 6 | 2 | 71169 | 15.664 | 1.305 | 0.001 |
| 6 | 3 | 6178 | 22.275 | 1.237 | 0.004 | 6 | 3 | 48673 | 22.904 | 1.272 | 0.002 |
| 6 | 4 | 4675 | 29.436 | 1.227 | 0.003 | 6 | 4 | 36146 | 30.841 | 1.285 | 0.001 |
| 7 | 1 | 15071 | 9.131 | 1.304 | 0.002 | 7 | 1 | 121097 | 9.206 | 1.315 | 0.000 |
| 7 | 2 | 7683 | 17.911 | 1.279 | 0.003 | 7 | 2 | 61071 | 18.254 | 1.304 | 0.001 |
| 7 | 3 | 5345 | 25.746 | 1.226 | 0.004 | 7 | 3 | 41774 | 26.686 | 1.271 | 0.002 |
| 7 | 4 | 4061 | 33.887 | 1.210 | 0.003 | 7 | 4 | 31067 | 35.884 | 1.282 | 0.001 |
| 8 | 1 | 13265 | 10.374 | 1.297 | 0.003 | 8 | 1 | 106028 | 10.514 | 1.314 | 0.000 |
| 8 | 2 | 6768 | 20.333 | 1.271 | 0.003 | 8 | 2 | 53461 | 20.852 | 1.303 | 0.001 |
| 8 | 3 | 4709 | 29.224 | 1.218 | 0.004 | 8 | 3 | 36575 | 30.480 | 1.270 | 0.002 |
| 8 | 4 | 3567 | 38.580 | 1.206 | 0.003 | 8 | 4 | 27238 | 40.928 | 1.279 | 0.001 |
| 9 | 1 | 11830 | 11.633 | 1.293 | 0.003 | 9 | 1 | 94321 | 11.819 | 1.313 | 0.000 |
| 9 | 2 | 6078 | 22.641 | 1.258 | 0.003 | 9 | 2 | 47572 | 23.434 | 1.302 | 0.001 |
| 9 | 3 | 4230 | 32.533 | 1.205 | 0.004 | 9 | 3 | 32577 | 34.220 | 1.267 | 0.002 |
| 9 | 4 | 3205 | 42.937 | 1.193 | 0.003 | 9 | 4 | 24278 | 45.918 | 1.275 | 0.001 |
| 10 | 1 | 10679 | 12.886 | 1.289 | 0.003 | 10 | 1 | 84938 | 13.125 | 1.312 | 0.000 |
| 10 | 2 | 5511 | 24.971 | 1.249 | 0.003 | 10 | 2 | 42884 | 25.996 | 1.300 | 0.001 |
| 10 | 3 | 3856 | 35.688 | 1.190 | 0.004 | 10 | 3 | 29345 | 37.989 | 1.266 | 0.001 |
| 10 | 4 | 2984 | 46.117 | 1.153 | 0.004 | 10 | 4 | 21876 | 50.960 | 1.274 | 0.001 |

*This page intentionally left blank*



# CHAPTER 34

# Partitioning and Broadcast, Part 2

in which we design a hybrid parallel program that requires broadcasting messages; we learn how to mingle message passing code with multithreaded code; we derive a mathematical model for the program's computation plus communication time; and we consider the implications for hybrid parallel program design

# CHAPTER 34 Partitioning and Broadcast, Part 2

## 34.1 Floyd's Algorithm on a Hybrid

Recall the **all-pairs shortest-paths problem**. We are given an input $n \times n$ distance matrix $d$ representing a graph with $n$ vertices, and we are to compute an output distance matrix giving the length of the shortest path between each pair of vertices using Floyd's Algorithm. The sequential version was the following:

```
for $i = 0$ to $n–1$
    for $r = 0$ to $n–1$
        for $c = 0$ to $n–1$
            $d_{rc} \leftarrow \min (d_{rc}, d_{ri} + d_{ic})$
```

When we designed the SMP parallel version in Chapter 16, we realized there was a sequential dependency from each outer loop iteration to the next. Accordingly, we left the outer loop as a sequential loop and made the middle loop a parallel loop. The parallel loop partitioned the distance matrix rows 0 through $n$–1 among the team threads.

```
for $i = 0$ to $n–1$
    parallel for $r = 0$ to $n–1$
        for $c = 0$ to $n–1$
            $d_{rc} \leftarrow \min (d_{rc}, d_{ri} + d_{ic})$
```

The cluster parallel version in Chapter 25 used a Range object to partition the distance matrix rows among the processes. Each process allocated storage only for its own slice of the distance matrix. Consequently, at the beginning of each outer loop iteration, the program had to broadcast row $i$ from the process that owned row $i$ to the other processes. Each process did all the outer loop iterations, a subrange of the middle loop iterations, and all the inner loop iterations.

```
$lb \leftarrow$ This process's row slice lower bound
$ub \leftarrow$ This process's row slice upper bound
Allocate storage for rows $lb$ to $ub$ of $d$
for $i = 0$ to $n–1$
    Broadcast row $i$ of $d$
    for $r = lb$ to $ub$
        for $c = 0$ to $n–1$
            $d_{rc} \leftarrow \min (d_{rc}, d_{ri} + d_{ic})$
```

The hybrid parallel version will be the same as the cluster version, except we'll use a parallel team to execute the middle loop iterations in multiple threads, like the SMP version. Thus, the computation is partitioned among the processes and threads in the same manner as the hybrid parallel AES key search program in Chapter 32.

```
$lb$ ← This process’s row slice lower bound
$ub$ ← This process’s row slice upper bound
Allocate storage for rows $lb$ to $ub$ of $d$
for $i = 0$ to $n–1$
   Broadcast row $i$ of $d$
   parallel for $r = lb$ to $ub$
      for $c = 0$ to $n–1$
         $d_{rc} \leftarrow \min(d_{rc}, d_{ri} + d_{ic})$
```

Thread coordination and synchronization is needed at three points in the hybrid version. We must make sure that only one thread in each process performs the broadcast at the top of the outer loop. If multiple threads tried to broadcast at the same time, there would be chaos. In addition, we must ensure that the threads do not start the middle parallel for loop until the broadcast has finished, and that the program does not go on to the next outer loop iteration until all the threads have finished the middle loop iterations.

Like the cluster parallel version, the hybrid parallel version uses the *parallel input files pattern* to reduce the time spent reading the input file. The hybrid parallel version also uses the *parallel output files pattern* to reduce the time spent writing the output file. The file I/O must be done by a single thread. Figure 34.1 shows the program’s overall execution timeline.

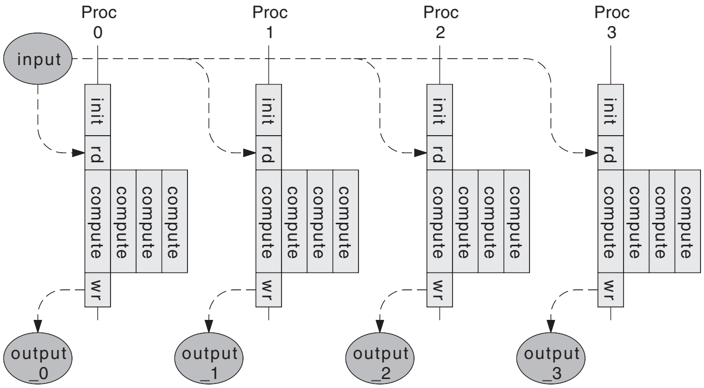


**Figure 34.1** Hybrid parallel program execution timeline

## 34.2 Hybrid Parallel Floyd’s Algorithm Program

Here is the code for the hybrid parallel version of the Floyd’s Algorithm program, class edu.rit.hyb.network. FloydHyb. It is mostly the same as the cluster parallel version, except for the parallel team in the middle.

```
package edu.rit.hyb.network;
import edu.rit.io.DoubleMatrixFile;
import edu.rit.io.Files;
import edu.rit.mp.DoubleBuf;
import edu.rit.pj.Comm;
import edu.rit.pj.IntegerForLoop;
import edu.rit.pj.ParallelRegion;
import edu.rit.pj.ParallelTeam;
import edu.rit.util.Range;
import java.io.BufferedInputStream;
import java.io.BufferedOutputStream;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
public class FloydHyb
   {
   // World communicator.
   static Comm world;
   static int size;
   static int rank;

   // Number of nodes.
   static int n;

   // Distance matrix.
   static double[][] d;

   // Slices of distance matrix.
   static Range[] ranges;
   static Range myrange;
   static int mylb;
   static int myub;

   // Row broadcast from another process.
   static double[] row_i;
   static DoubleBuf row_i_buf;

   // Outer loop index i.
   static int i;

   // Reference to row i.
   static double[] d_i;
```

The program starts out running in a single thread—the main program thread. The initialization and reading of the input file are done in this one thread.

```
   /**
    * Main program.
    */
   public static void main
      (String[] args)
      throws Throwable
      {
      // Start timing.
      long t1 = System.currentTimeMillis();

      // Initialize world communicator.
      Comm.init (args);
      world = Comm.world();
      size = world.size();
      rank = world.rank();

      // Parse command line arguments.
      if (args.length != 2) usage();
      File infile = new File (args[0]);
      File outfile = new File (args[1]);

      // Prepare to read distance matrix from input file; determine
      // matrix dimensions.
      DoubleMatrixFile in = new DoubleMatrixFile();
      DoubleMatrixFile.Reader reader =
         in.prepareToRead
            (new BufferedInputStream
               (new FileInputStream (infile)));
      d = in.getMatrix();
      n = d.length;

      // Divide distance matrix into equal row slices.
      ranges = new Range (0, n-1) .subranges (size);
      myrange = ranges[rank];
      mylb = myrange.lb();
      myub = myrange.ub();

      // Read just this process's row slice of the distance matrix.
      reader.readRowSlice (myrange);
      reader.close();

      // Allocate storage for row broadcast from another process.
      row_i = new double [n];
      row_i_buf = DoubleBuf.buffer (row_i);

      long t2 = System.currentTimeMillis();

      // Run Floyd's Algorithm.
      //     for i = 0 to N-1
      //         for r = 0 to N-1
      //             for c = 0 to N-1
      //                 D[r,c] = min (D[r,c], D[r,i] + D[i,c])
      int i_root = 0;
```

Now comes the computation section. We create a parallel team for later use in the middle loop. Because we don't want to create new threads continually, we create the parallel team just once, before beginning the outer loop. However, the parallel team does not execute any code yet. Rather, the one main program thread does the outer loop iterations and performs the broadcasts of row *i*.

```
      ParallelTeam team = new ParallelTeam();
      for (i = 0; i < n; ++ i)
         {
         d_i = d[i];

         // Determine which process owns row i.
         if (! ranges[i_root].contains (i)) ++ i_root;

         // Broadcast row i from owner process to all processes.
         if (rank == i_root)
            {
            world.broadcast (i_root, DoubleBuf.buffer (d_i));
            }
         else
            {
            world.broadcast (i_root, row_i_buf);
            d_i = row_i;
            }
```

We've arrived at the middle loop, to be executed in parallel. We tell the parallel team to execute the parallel for loop. (We use the same parallel team each time through the outer loop.) Because the broadcast was performed by a single thread, the middle loop iterations do not begin until after the broadcast has finished and the `broadcast()` method has returned. Because the team threads wait for each other at a barrier at the end of the parallel for loop, the next outer loop iteration does not begin until all the threads have finished the middle loop iterations.

```
         // Inner loops over rows in my slice and over all columns.
         team.execute (new ParallelRegion()
            {
            public void run() throws Exception
               {
               execute (mylb, myub, new IntegerForLoop()
                  {
```

```
                  public void run (int first, int last)
                     throws Exception
                     {
                     for (int r = first; r <= last; ++ r)
                        {
                        double[] d_r = d[r];
                        for (int c = 0; c < n; ++ c)
                           {
                           d_r[c] = Math.min(d_r[c],d_r[i]+d_i[c]);
                           }
                        }
                     }
                  });
               }
            });
         }

      long t3 = System.currentTimeMillis();
```

After the computations, we're back in the single main program thread to write the output file and finish the program.

```
      // Write distance matrix slice to a separate output file in
      // each process.
      DoubleMatrixFile out = new DoubleMatrixFile (n, n, d);
      DoubleMatrixFile.Writer writer =
         out.prepareToWrite
            (new BufferedOutputStream
               (new FileOutputStream
                  (Files.fileForRank (outfile, rank))));
      writer.writeRowSlice (myrange);
      writer.close();

      // Stop timing.
      long t4 = System.currentTimeMillis();
      System.out.println ((t2-t1) + " msec pre " + rank);
      System.out.println ((t3-t2) + " msec calc " + rank);
      System.out.println ((t4-t3) + " msec post " + rank);
      System.out.println ((t4-t1) + " msec total " + rank);
      }
   }
```

## 34.3 Computation-Time Model

To gain some insight into how the hybrid parallel version's performance differs from the cluster parallel version, let's derive a computation-time model. Instead of depending on the number of vertices *n* and the number of processors *K*, the model depends on *n*, the number of processes *Kp*, and the number of threads per process *Kt*.

The hybrid version's calculation time is the same as the cluster version (Equation 25.2), substituting *Kp* · *Kt* for *K*:

$$T_{\text{calc}}(n, Kp, Kt) = \frac{8.86 \times 10^{-9} n^3}{Kp \cdot Kt} \tag{34.1}$$

The hybrid version does the same broadcasts as the cluster version (Equation 25.3), except the messages only go to *Kp* processes rather than *K* processors. The hybrid version's broadcast time therefore is the following:

$$T_{\text{bcast}}(n, Kp, Kt) = n(2.08 \times 10^{-4} + 6.85 \times 10^{-8} n) \cdot \text{ceil}(\log_2 Kp) \tag{34.2}$$

Adding Equations 34.1 and 34.2 together gives the computation-time model for the hybrid version:

$$\begin{aligned} T(n, Kp, Kt) = {} & \frac{8.86 \times 10^{-9} n^3}{Kp \cdot Kt} \\ & + n(2.08 \times 10^{-4} + 6.85 \times 10^{-8} n) \cdot \text{ceil}(\log_2 Kp) \end{aligned} \tag{34.3}$$

The chief difference between the cluster version's computation-time model and the hybrid version's computation-time model is that in the latter, the broadcast time increases only as the logarithm of the number of *processes,* not the logarithm of the number of processors. Thus, the hybrid version has less message passing overhead than the cluster version. For example, on the 40 processors of the "tardis" computer, the cluster version requires six message rounds to do a broadcast, but the hybrid version requires only four message rounds—33 percent less message passing. This translates into reduced running times and better performance for the hybrid version.

However, the hybrid version's computation-time model still has a term that increases as the number of processes increases. Thus, the hybrid version will still experience a slowdown above a certain number of processes, and we must be careful not to run the hybrid version on too many nodes of the hybrid parallel computer.

## 34.4 Hybrid Floyd's Algorithm Performance

Table 34.1 (at the end of the chapter) lists, and Figure 34.2 plots, the FloydHyb program's performance on the "tardis" parallel computer. The running times are for the calculation portion only, not including the distance matrix file I/O. The program was run on two problem sizes: $n = 3{,}180$ vertices ($N = n^3 = 32\text{G}$) and $n = 6{,}360$ vertices ($N = 256\text{G}$).

Comparing the data to Chapter 25, most of the hybrid parallel version's running times are smaller than the cluster parallel version's. This is partially because of the reduced number of messages needed to do the broadcasts, as the computation time model predicts. The reduced running times are also due to the effect we saw with the SMP parallel version in Chapter 16. As we go to more threads on each node, the slice of the distance matrix accessed by each thread becomes smaller, more of the distance matrix slice fits in the CPU's cache memory, the number of cache line replacements goes down, and the program's speed goes up relative to the sequential version.

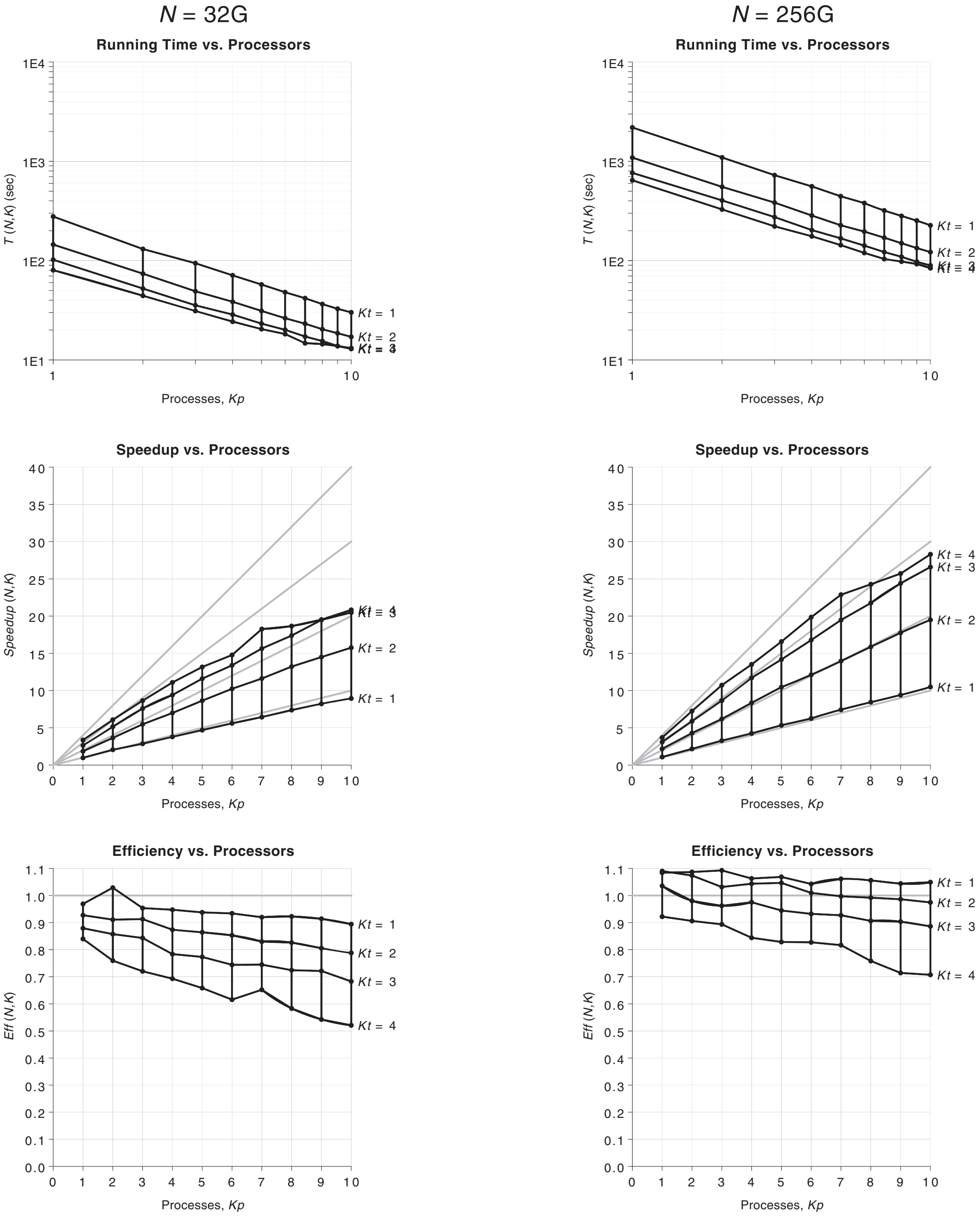


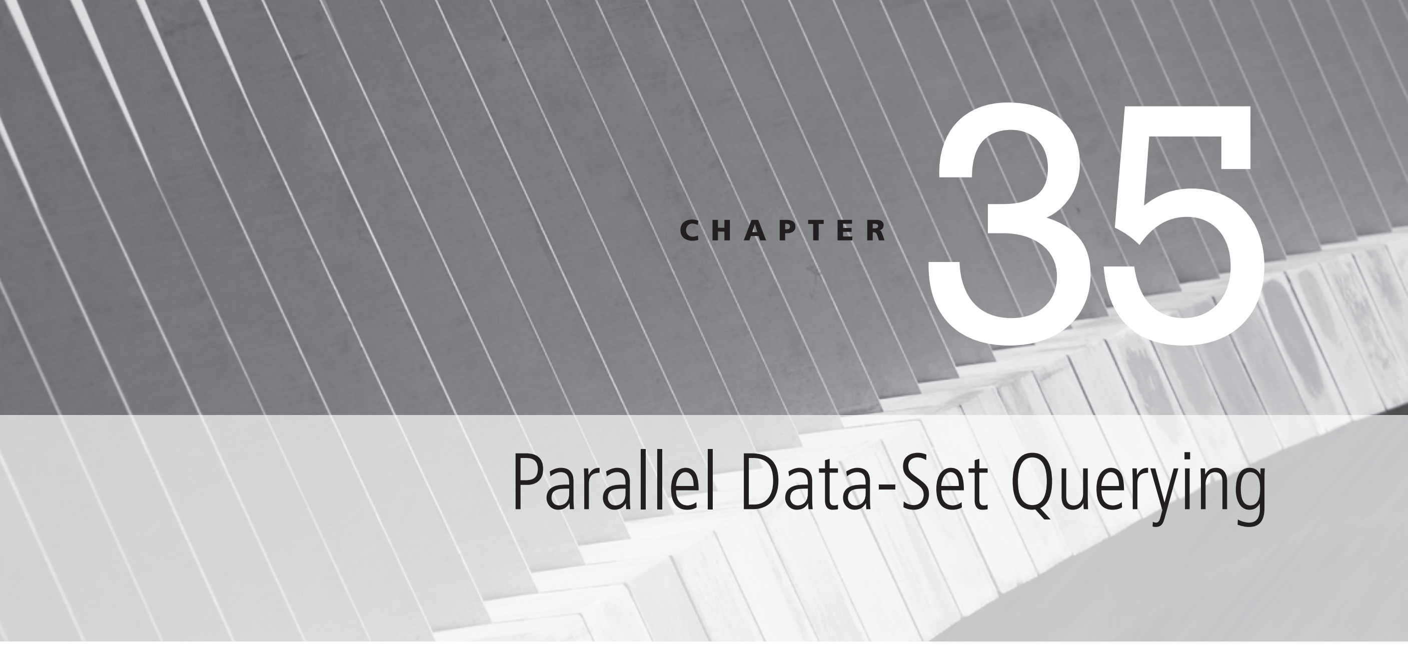
**Figure 34.2** FloydSeq/FloydHyb running-time metrics

Despite the program's cache behavior, the computation time model predicts that eventually the speedups will flatten out and then decrease as the number of processors increases, due to the increasing communication time. This effect is apparent in the speedup curves, which are far from ideal on larger numbers of processes. In fact, with 10 processes, the program is not much faster running on four threads per process than three threads per process. While Floyd's Algorithm does somewhat better on a hybrid parallel computer than on a cluster parallel computer, the program still has too much message passing to scale up well.

**Table 34.1** FloydSeq/FloydHyb running-time metrics

| *N* = 32G | | | | | | *N* = 256G | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| *Kp* | *Kt* | *T* | *Spdup* | *Eff* | *EDSF* | *Kp* | *Kt* | *T* | *Spdup* | *Eff* | *EDSF* |
| seq | seq | 269235 | | | | seq | seq | 2375802 | | | |
| 1 | 1 | 277854 | 0.969 | 0.969 | | 1 | 1 | 2191148 | 1.084 | 1.084 | |
| 1 | 2 | 145158 | 1.855 | 0.927 | 0.045 | 1 | 2 | 1089687 | 2.180 | 1.090 | -0.005 |
| 1 | 3 | 102073 | 2.638 | 0.879 | 0.051 | 1 | 3 | 764827 | 3.106 | 1.035 | 0.024 |
| 1 | 4 | 80152 | 3.359 | 0.840 | 0.051 | 1 | 4 | 644102 | 3.689 | 0.922 | 0.059 |
| 2 | 1 | 130833 | 2.058 | 1.029 | -0.058 | 2 | 1 | 1093020 | 2.174 | 1.087 | -0.002 |
| 2 | 2 | 73890 | 3.644 | 0.911 | 0.021 | 2 | 2 | 553076 | 4.296 | 1.074 | 0.003 |
| 2 | 3 | 52328 | 5.145 | 0.858 | 0.026 | 2 | 3 | 403962 | 5.881 | 0.980 | 0.021 |
| 2 | 4 | 44303 | 6.077 | 0.760 | 0.039 | 2 | 4 | 327721 | 7.249 | 0.906 | 0.028 |
| 3 | 1 | 94110 | 2.861 | 0.954 | 0.008 | 3 | 1 | 724724 | 3.278 | 1.093 | -0.004 |
| 3 | 2 | 49152 | 5.478 | 0.913 | 0.012 | 3 | 2 | 383887 | 6.189 | 1.031 | 0.010 |
| 3 | 3 | 35478 | 7.589 | 0.843 | 0.019 | 3 | 3 | 274392 | 8.658 | 0.962 | 0.016 |
| 3 | 4 | 31148 | 8.644 | 0.720 | 0.031 | 3 | 4 | 221541 | 10.724 | 0.894 | 0.019 |
| 4 | 1 | 71035 | 3.790 | 0.948 | 0.008 | 4 | 1 | 558817 | 4.251 | 1.063 | 0.007 |
| 4 | 2 | 38513 | 6.991 | 0.874 | 0.016 | 4 | 2 | 284569 | 8.349 | 1.044 | 0.006 |
| 4 | 3 | 28624 | 9.406 | 0.784 | 0.021 | 4 | 3 | 203019 | 11.702 | 0.975 | 0.010 |
| 4 | 4 | 24282 | 11.088 | 0.693 | 0.027 | 4 | 4 | 175907 | 13.506 | 0.844 | 0.019 |
| 5 | 1 | 57422 | 4.689 | 0.938 | 0.008 | 5 | 1 | 444609 | 5.344 | 1.069 | 0.004 |
| 5 | 2 | 31153 | 8.642 | 0.864 | 0.013 | 5 | 2 | 226994 | 10.466 | 1.047 | 0.004 |
| 5 | 3 | 23212 | 11.599 | 0.773 | 0.018 | 5 | 3 | 167650 | 14.171 | 0.945 | 0.011 |
| 5 | 4 | 20449 | 13.166 | 0.658 | 0.025 | 5 | 4 | 143370 | 16.571 | 0.829 | 0.016 |
| 6 | 1 | 48042 | 5.604 | 0.934 | 0.007 | 6 | 1 | 379760 | 6.256 | 1.043 | 0.008 |
| 6 | 2 | 26304 | 10.236 | 0.853 | 0.012 | 6 | 2 | 196074 | 12.117 | 1.010 | 0.007 |
| 6 | 3 | 20099 | 13.395 | 0.744 | 0.018 | 6 | 3 | 141571 | 16.782 | 0.932 | 0.010 |
| 6 | 4 | 18223 | 14.774 | 0.616 | 0.025 | 6 | 4 | 119661 | 19.854 | 0.827 | 0.014 |
| 7 | 1 | 41813 | 6.439 | 0.920 | 0.009 | 7 | 1 | 319794 | 7.429 | 1.061 | 0.004 |
| 7 | 2 | 23173 | 11.618 | 0.830 | 0.013 | 7 | 2 | 170173 | 13.961 | 0.997 | 0.007 |
| 7 | 3 | 17212 | 15.642 | 0.745 | 0.015 | 7 | 3 | 122031 | 19.469 | 0.927 | 0.008 |
| 7 | 4 | 14746 | 18.258 | 0.652 | 0.018 | 7 | 4 | 103875 | 22.872 | 0.817 | 0.012 |
| 8 | 1 | 36472 | 7.382 | 0.923 | 0.007 | 8 | 1 | 281256 | 8.447 | 1.056 | 0.004 |
| 8 | 2 | 20367 | 13.219 | 0.826 | 0.012 | 8 | 2 | 149683 | 15.872 | 0.992 | 0.006 |
| 8 | 3 | 15488 | 17.383 | 0.724 | 0.015 | 8 | 3 | 109221 | 21.752 | 0.906 | 0.009 |
| 8 | 4 | 14432 | 18.655 | 0.583 | 0.021 | 8 | 4 | 97899 | 24.268 | 0.758 | 0.014 |
| 9 | 1 | 32729 | 8.226 | 0.914 | 0.008 | 9 | 1 | 252813 | 9.397 | 1.044 | 0.005 |
| 9 | 2 | 18570 | 14.498 | 0.805 | 0.012 | 9 | 2 | 133828 | 17.753 | 0.986 | 0.006 |
| 9 | 3 | 13821 | 19.480 | 0.721 | 0.013 | 9 | 3 | 97383 | 24.396 | 0.904 | 0.008 |
| 9 | 4 | 13789 | 19.525 | 0.542 | 0.022 | 9 | 4 | 92406 | 25.710 | 0.714 | 0.015 |
| 10 | 1 | 30107 | 8.943 | 0.894 | 0.009 | 10 | 1 | 226459 | 10.491 | 1.049 | 0.004 |
| 10 | 2 | 17082 | 15.761 | 0.788 | 0.012 | 10 | 2 | 121887 | 19.492 | 0.975 | 0.006 |
| 10 | 3 | 13144 | 20.483 | 0.683 | 0.014 | 10 | 3 | 89356 | 26.588 | 0.886 | 0.008 |
| 10 | 4 | 12923 | 20.834 | 0.521 | 0.022 | 10 | 4 | 83968 | 28.294 | 0.707 | 0.014 |

*This page intentionally left blank*



# CHAPTER 35

# Parallel Data-Set Querying

in which we are introduced to problems that involve extracting answers from massive data sets; we examine strategies for solving such problems on a parallel computer; and we design a hybrid parallel program to query a data set in the domain of number theory

# CHAPTER 35 Parallel Data-Set Querying

## 35.1 Data Sets and Queries

All the parallel programs we've studied so far have been designed to solve an instance of some problem from scratch. The problem's parameters have been specified as command-line arguments. In some cases, input data for the problem came from a file, such as the distance matrix file for Floyd's Algorithm. The command-line arguments and input files were unique for that instance of the problem.

However, some users are interested in a different kind of problem. There is a preexisting set of data; the data could consist of experimental measurements, or the data could be the result of a prior computation. The user wants to answer some question by analyzing the data. We will use the term **data set** to refer to this preexisting data. The data set may be stored in a file, or in a group of files, or even in a traditional database. We will call the question a **query**. The query may involve simply looking up some information in the data set, or the query may require extensive computations on the contents of the data set. The process of finding the answer is **data-set querying**. Rather than solving different, unique instances of a problem, a data-set querying program answers different questions about the *same* data set.

Here's an example of a data-set querying problem, from the domain of computational biology. The biologist has a *data set* of protein sequences, including information about each protein's function. The biologist also has a *query,* which is the sequence of an unknown protein. The data-set querying problem is to find the protein or proteins in the data set that most closely match the query. The matching known proteins then yield clues to the unknown protein's function. Solving this data-set querying problem requires lengthy computations to "align" the query protein sequence with each protein sequence in the data set. The Basic Local Alignment Search Tool (BLAST) is a program widely used to solve this data-set querying problem for protein and DNA sequences.

## 35.2 Parallel Data-Set Querying Strategies

When the data set is large, or when the query computations are lengthy, or both, a parallel program can speed up solving the data-set querying problem. To get a speedup, the problem must be partitioned into pieces that can be computed in parallel. There are two basic strategies for partitioning a data-set querying problem.

The first strategy is applicable when the data set is relatively small—small enough to fit completely in the memory of each parallel computer node. In this case, we can **replicate the data set** in each node and **partition the query** computations among the nodes (Figure 35.1). Each node performs part of the query computations using the entire data set. The nodes then use reduction of some kind to combine their partial answers into the overall answer.

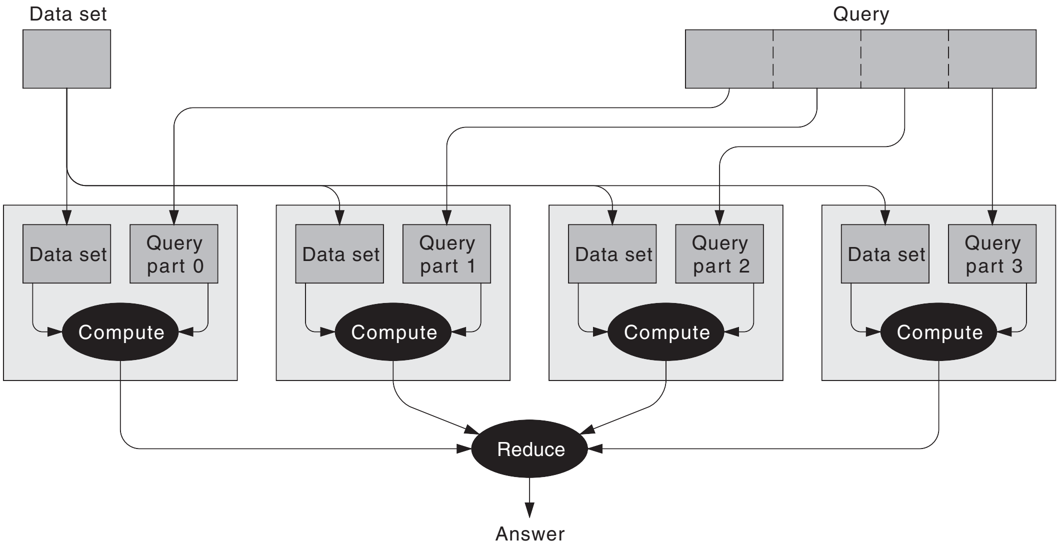


**Figure 35.1** Parallel data-set querying strategy 1—replicate the data set, partition the query

The second strategy is applicable when the data set is too large to fit in the memory of one node. In this case, we can **partition the data set** among the nodes and **replicate the query** in each node (Figure 35.2). Each node performs all of the query computations on its own portion of the data set. Again, the nodes use reduction to get the overall answer.

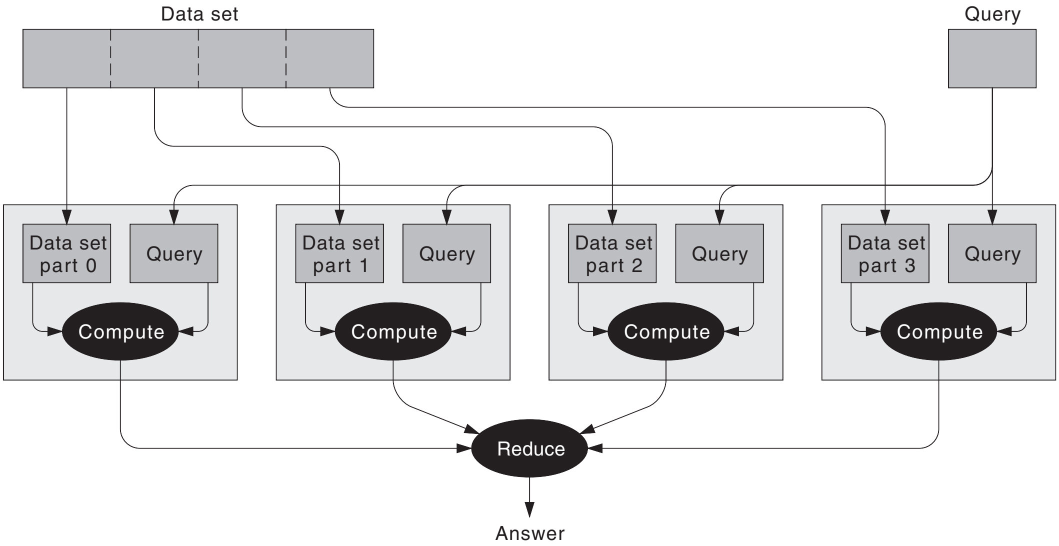


**Figure 35.2** Parallel data-set querying strategy 2—partition the data set, replicate the query

In this chapter, we will look at an example of the first parallel data-set querying strategy. In Chapter 37, we will look at an example of the second strategy.

## 35.3 The Prime Counting Function

A **prime** is an integer whose only factors are itself and 1. The first few primes are 2, 3, 5, 7, 11, 13, 17, 19, 23, and 29. (0 and 1 are not considered to be prime.) All primes except 2 are odd numbers. A number that is not a prime is called **composite**.

The **prime counting function** $\pi(x)$ is the number of primes less than or equal to $x$. This function has nothing to do with the constant $\pi$, the ratio of a circle's circumference to its diameter. If you're confused, blame number theorist Edmund Landau, who invented this now-standard notation for the prime counting function in 1909. The prime counting function itself has been studied since antiquity.

The prime counting function touches on some of today's most widely used computer applications. Every time your Web browser sets up an HTTPS connection to a secure Web site, it's likely that the **RSA public key cryptosystem** is being used to protect your credit card number from prying eyes on the Internet. Invented by Ronald Rivest, Adi Shamir, and Leonard Adleman in 1978, the RSA algorithm uses large primes. How large? Cryptographers Niels Ferguson and Bruce Schneier recommend that RSA primes should be at least 1,024 bits long when expressed in binary. That is, an RSA prime $x$ should fall in the range $2^{1,023} < x < 2^{1,024}$. How many primes are there in this range? If there are only a few, then RSA won't be very useful. But if there are many, there will be plenty of primes to let everyone in the world have their own unique RSA public keys. The number of primes in this range is $\pi(2^{1,024}) - \pi(2^{1,023})$. Hence, we are interested in evaluating $\pi(x)$.

The **prime number theorem** gives an estimate for $\pi(x)$. Many variations of the prime number theorem have been proven over the years. One version says that $\pi(x)$ is approximately $x/(\ln x)$. Using that formula, the number of 1,024-bit primes is approximately $1.27 \times 10^{305}$. We are in no danger of running out of RSA primes any time soon.

To illustrate parallel data set querying techniques, we will design a program to find the *exact* value of $\pi(x)$, not just an estimate. To do that, we simply will list all the primes less than or equal to $x$, and then count them.

## 35.4 Sieving

To list primes, we use the **Sieve of Eratosthenes**. A Greek mathematician and astronomer who lived in the third century BCE, Eratosthenes was the first person to calculate the circumference of the earth. He also invented the sieving algorithm for listing prime numbers that bears his name. While modern sieving algorithms are faster, Eratosthenes's sieve is still the simplest.

An example illustrates how Eratosthenes's sieve works. Let's say we want to find all the primes from 100 to 149. We start by making a **sieve**, which is a list of all those numbers.

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 100 | 101 | 102 | 103 | 104 | 105 | 106 | 107 | 108 | 109 |
| 110 | 111 | 112 | 113 | 114 | 115 | 116 | 117 | 118 | 119 |
| 120 | 121 | 122 | 123 | 124 | 125 | 126 | 127 | 128 | 129 |
| 130 | 131 | 132 | 133 | 134 | 135 | 136 | 137 | 138 | 139 |
| 140 | 141 | 142 | 143 | 144 | 145 | 146 | 147 | 148 | 149 |

Next, we need a list of primes, in ascending order. (For the moment, we won't worry about where this list comes from.) We take the first prime from the list of primes, 2, and remove all multiples of 2 from the sieve, starting with the first multiple of 2 greater than or equal to 100. These numbers cannot be prime, because they have 2 as a factor.

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | 101 | | 103 | | 105 | | 107 | | 109 |
| | 111 | | 113 | | 115 | | 117 | | 119 |
| | 121 | | 123 | | 125 | | 127 | | 129 |
| | 131 | | 133 | | 135 | | 137 | | 139 |
| | 141 | | 143 | | 145 | | 147 | | 149 |

We take the next prime from the list of primes, 3, and remove all multiples of 3 from the sieve, starting with the first multiple of 3 greater than or equal to 100. However, notice we can save a bit of work by striking out only the *odd* multiples of 3. (The even multiples were already removed.) The first odd multiple of 3 greater than or equal to 100 is 105, and to get successive odd multiples of 3, we go up in steps of 6: 105, 111, 117, 123, 129, 135, 141, 147.

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | 101 | | 103 | | | | 107 | | 109 |
| | | | 113 | | 115 | | | | 119 |
| | 121 | | | | 125 | | 127 | | |
| | 131 | | 133 | | | | 137 | | 139 |
| | | | 143 | | 145 | | | | 149 |

For the next prime, 5, we remove 105, 115, 125, 135, and 145. (Some of these were already removed.)

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | 101 | | 103 | | | | 107 | | 109 |
| | | | 113 | | | | | | 119 |
| | 121 | | | | | | 127 | | |
| | 131 | | 133 | | | | 137 | | 139 |
| | | | 143 | | | | | | 149 |

For the next prime, 7, we remove 105, 119, 133, and 147.

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | 101 | | 103 | | | | 107 | | 109 |
| | | | 113 | | | | | | |
| | 121 | | | | | | 127 | | |
| | 131 | | | | | | 137 | | 139 |
| | | | 143 | | | | | | 149 |

For the next prime, 11, we remove 121 and 143.

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | 101 | | 103 | | | | 107 | | 109 |
| | | | 113 | | | | | | |
| | | | | | | | 127 | | |
| | 131 | | | | | | 137 | | 139 |
| | | | | | | | | | 149 |

For the next prime, 13, we normally would start removing numbers at $9 \cdot 13$, or 117. However, we already removed all multiples of 9, back when we removed multiples of 3. In fact, all multiples of 13 less than $13 \cdot 13$ have already been removed. So we can start removing numbers at $13^2$, or 169. But 169 is past the end of the sieve, so we're done. The numbers left in the sieve are the primes falling between the original bounds (100 to 149).

Here is the source code for class edu.rit.hyb.prime.Sieve, which encapsulates the preceding algorithm. A Sieve object operates on a block of numbers of a given length `(len)` starting at a given lower bound `(lb)`. The sieve itself is stored as an array of `boolean` flags; to put a number in the sieve or remove a number from the sieve, just set the corresponding flag true or false.

```
package edu.rit.hyb.prime;
import java.io.IOException;
public class Sieve
   {
   private boolean[] isPrime;
   private long lb;
   private int len;

   // Padding to avert cache interference.
   private long p0, p1, p2, p3, p4, p5, p6, p7;
   private long p8, p9, pa, pb, pc, pd, pe, pf;

   /**
    * Construct a new sieve object.
    *
    * @param  lb   Lower bound index. Assumed to be a nonnegative
    *              even number.
    * @param  len  Length. Assumed to be a nonnegative even number.
    */
   public Sieve
      (long lb,
       int len)
      {
      this.isPrime = new boolean [len+128]; // Padding
      this.lb = lb;
      this.len = len;
      }

   /**
    * Get this sieve's lower bound index.
    *
    * @return  Lower bound index.
    */
   public long lb()
      {
```

```
      return this.lb;
      }

   /**
    * Set this sieve’s lower bound index.
    *
    * @param  lb  Lower bound index. Assumed to be a nonnegative
    *             even number.
    */
   public void lb
      (long lb)
      {
      this.lb = lb;
      }

   /**
    * Get this sieve’s length.
    *
    * @return  Length.
    */
   public long length()
      {
      return this.len;
      }

   /**
    * Set this sieve’s length.
    *
    * @param  len  Length. Assumed to be a nonnegative even number.
    */
   public void len
      (int len)
      {
      this.isPrime = new boolean [len+128]; // Padding
      this.len = len;
      }

   /**
    * Initialize this sieve. Afterwards, all even-numbered flags are
    * false and all odd-numbered flags are true.
    */
   public void initialize()
      {
      for (int i = 1; i < len; i += 2) isPrime[i] = true;
      if (lb == 0) isPrime[1] = false; // 1 is not a prime
      }
```

```
/**
 * Sieve out the given prime. Afterwards, all flags corresponding
 * to multiples of the given prime are false. It is assumed that
 * p is an odd prime and that all multiples of smaller primes
 * have previously been sieved out.
 *
 * @param  p  Prime to sieve out.
 *
 * @return  True if sieving with further primes is required,
 *          false otherwise.
 */
public boolean sieveOut
   (long p)
   {
   // If p^2 is beyond the end of the array, report that further
   // sieving is not required.
   long psqr = p*p;
   if (psqr - lb >= len) return false;

   // Find the first odd multiple of p greater than or equal to
   // lb.
   long m = (lb + p - 1) / p;
   if ((m & 1) == 0) ++ m;
   long mp = m*p;

   // Sieving begins at p^2 or mp, whichever is larger.
   mp = Math.max (mp, psqr);

   // Set all odd multiples of p to false and report that
   // further sieving is required.
   long two_p = 2*p;
   for (long i = mp - lb; i < len; i += two_p)
      {
      isPrime[(int) i] = false;
      }
   return true;
   }

/**
 * Sieve out all primes returned by the given iterator. It is
 * assumed that the iterator returns a sequence of the odd
 * primes in ascending order (3, 5, 7, 11, . . .). Sieving
 * continues until no further sieving is required or until the
 * end of the iterator’s sequence of primes, whichever comes
 * first.
 *
 * @param  iterator  Iterator for a sequence of odd primes.
 *
 * @exception  IOException
 *     Thrown if an I/O error occurred.
 */
public void sieveOut
   (LongIterator iterator)
   throws IOException
   {
   long p;
   initialize();
   while ((p = iterator.next()) != 0 && sieveOut (p));
   }

/**
 * Determine if the given number is prime. It is assumed that all
 * primes smaller than p have been sieved out.
 *
 * @param  p  Number to test.
 *
 * @return  True if p is prime, false otherwise.
 */
public boolean isPrime
   (long p)
   {
   return isPrime[(int)(p - lb)];
   }

/**
 * Obtain an iterator for the primes in this sieve. The iterator
 * returns a sequence of the numbers whose flags are true in this
 * sieve.
 *
 * @return  Iterator.
 */
public LongIterator iterator()
   {
   return new LongIterator()
      {
      private int i = 0;

      // Padding to avert cache interference.
      private long p0, p1, p2, p3, p4, p5, p6, p7;
      private long p8, p9, pa, pb, pc, pd, pe, pf;
```

```
         public long next()
            {
            do ++ i; while (i < len && ! isPrime[i]);
            return i < len ? lb + i : 0;
            }

         public void close()
            {
            }
         };
      }
   }
```

Class Sieve's `iterator()` method returns an iterator that returns a sequence of numbers, which are the primes in the sieve. The iterator implements interface LongIterator.

```
package edu.rit.hyb.prime;
import java.io.IOException;
public interface LongIterator
   {
   /**
    * Returns the next number in the sequence.
    *
    * @return  Number, or 0 if there are no more numbers.
    *
    * @exception  IOException
    *     Thrown if an I/O error occurred.
    */
   public long next()
      throws IOException;

   /**
    * Close this iterator. Call close() when done using this
    * iterator, to release resources.
    *
    * @exception  IOException
    *     Thrown if an I/O error occurred.
    */
   public void close()
      throws IOException;
   }
```

Given the Sieve class, we can cast the prime counting function as a data-set querying problem. The data set is a list of odd primes. The query is to count the number of primes less than or equal to some number *x*. The calculation is simply the following:

```
Create a sieve starting at 0 of length $x$
Initialize the sieve (i.e., remove all even numbers)
For each prime $p$ in the list of odd primes, until $p^2 > x$:
    Remove multiples of $p$ from the sieve
Count the primes left in the sieve
```

Note that because sieving stops when $p^2 > x$, the list of odd primes only has to go up to the square root of *x*. For example, if the list contains all odd primes less than $2^{32}$, we can calculate $\pi(x)$ for *x* up to $2^{64}$; that is, for all values *x* of type `long`. We can use the same data set to calculate $\pi(x)$ for any value of *x*.

However, there's a problem. A sieve of length *x* requires *x* bytes of storage to hold the array of `booleans`. Therefore, the computer's main memory size puts a limit on the length of the sieve we can compute.

To overcome this problem and calculate $\pi(x)$ for *x* larger than the computer's main memory, we can sieve the range of numbers *in multiple chunks* that are small enough to fit in memory. Suppose we choose a chunk size of one million numbers. We will count the primes from 0–999,999, from 1,000,000–1,999,999, and so on, and total the counts:

```
$L \leftarrow 1{,}000{,}000$
$count \leftarrow 0$
For $lb = 0$ to $x$ in steps of $L$:
    Create a sieve starting at $lb$ of length $L$
    Initialize the sieve (i.e., remove all even numbers)
    For each prime $p$ in the list of odd primes, until $p^2 > lb + L$:
            Remove multiples of $p$ from the sieve
    $count \leftarrow count$ + number of primes $\le x$ in the sieve
```

The only question remaining is where the list of odd primes comes from. We can use the same sieve algorithm, slightly modified, to find the odd primes. After initializing the sieve, the smallest odd number in the sieve is 3, so that is the first odd prime. After removing all multiples of 3 from the sieve, the smallest number in the sieve greater than 3 is 5, so that is the second odd prime. After removing all multiples of 5 from the sieve, the smallest number in the sieve greater than 5 is 7, so that is the third odd prime. Continuing this process, we enumerate all the odd primes.

We would like to have a data set with all the odd primes less than $2^{32}$. There happen to be 203,280,220 of them. If we stored each of them in a file as an eight-byte `long`, the file would occupy over 1.5 gigabytes. We can make the file smaller—and reduce the time the prime counting program takes to read it—by storing the *differences* between consecutive odd primes, rather than the primes themselves. Among the odd primes less than $2^{32}$, the largest difference is 336; it occurs in two places, between 3,842,610,773 and 3,842,611,109 and between 4,275,912,661 and 4,275,912,997. Furthermore, because the primes are all odd numbers, the differences are all even numbers, and we can save a bit more by storing the differences divided by 2 instead of the actual differences. The largest half-difference is 168; thus, each half-difference can be stored in a single byte, and the file occupies only 194 megabytes instead of 1.5 gigabytes.

Class edu.rit.hyb.prime.Prime32File in the Parallel Java Library uses sieving to find all the odd primes less than $2^{32}$ and store the half-differences between consecutive odd primes in a file. Because this program runs fairly quickly—it takes only 60 seconds on one of the "tardis" computer's nodes—it's not worth the effort to make it a parallel program. Class edu.rit.hyb.prime.Prime32List provides an object that reads the prime data set (file) and creates a LongIterator that returns the sequence of odd primes.

## 35.5 Sequential Prime Counting Program

Putting it all together, here is the source code for class edu.rit.hyb.prime.PrimeCountFunctionSeq, a sequential program that computes $\pi(x)$. It uses class Prime32List to get the list of odd primes and class Sieve to do the sieving.

```
package edu.rit.hyb.prime;
import java.io.File;
public class PrimeCountFunctionSeq
   {
   // Sieve in one-million-number chunks.
   static final int CHUNK = 1000000;

   // Command line arguments.
   static long x;
   static File primefile;

   /**
    * Main program.
    */
   public static void main
      (String[] args)
      throws Exception
      {
      // Start timing.
      long t1 = System.currentTimeMillis();

      // Parse command line arguments.
      if (args.length != 2) usage();
      x = Long.parseLong (args[0]);
      if (x < 0) usage();
      primefile = new File (args[1]);

      // Set up sieve.
      Sieve sieve = new Sieve (0, CHUNK);

      // Set up list of 32-bit primes.
      Prime32List primeList = new Prime32List (primefile);
      // For counting primes. Initially 1 to count prime number 2.
      long primeCount = 1;
```

Each time we sieve a new chunk of numbers, we create a new iterator for the data set of odd primes. The new iterator rereads the prime file from the beginning. This is not as inefficient as it sounds. Most operating systems will save the contents of files read from disk in a kernel cache in main memory. If the same file is reread, the contents will come from the kernel cache, taking much less time than rereading the contents from disk.

```
      // Do all chunks.
      for (long lb = 0; lb >= 0 && lb <= x; lb += CHUNK)
         {
         // Get an iterator for the odd primes.
         LongIterator iter = primeList.iterator();

         // Sieve the chunk.
         sieve.lb (lb);
         sieve.initialize();
         sieve.sieveOut (iter);

         // Count primes <= x left in the chunk.
         iter = sieve.iterator();
         long p;
         while ((p = iter.next()) != 0 && p <= x) ++ primeCount;
         }

      // Stop timing.
      long t2 = System.currentTimeMillis();

      // Print the answer.
      System.out.println ("pi("+x+") = "+primeCount);
      System.out.println ((t2-t1)+" msec");
      }
   }
```

## 35.6 Hybrid Parallel Prime Counting Program

To design a parallel version of the prime counting program, we must first decide which parallel data-set querying strategy to use. Although the data set occupies 194 megabytes, the prime counting program typically needs to read only a fraction of it, up to the square root of *x*. However, the program has to calculate sieves all the way up to *x*. Therefore, the first querying strategy makes the most sense: *replicate the data set*—each parallel processor reads the whole prime file; *partition the query*—the one-million-number sieves are divided among the parallel processors. Each processor counts the primes for a subset of the sieves, and at the end, the processors do a reduction to add all the counts together.

The next decision is how to partition the sieves among the processors. Sieves containing larger numbers take longer to calculate, because the program has to go through more primes *p* until $p^2$ is past the end of the sieve. Therefore, load balancing is required. We use the same *master-worker pattern with two-level scheduling* as we used for the hybrid parallel Mandelbrot Set program in Chapter 33.

Here is the source code for class edu.rit.hyb.prime.PrimeCountFunctionHyb, the hybrid parallel version.

```
package edu.rit.hyb.prime;
import edu.rit.mp.IntegerBuf;
import edu.rit.mp.LongBuf;
import edu.rit.mp.ObjectBuf;
import edu.rit.mp.buf.LongItemBuf;
import edu.rit.mp.buf.ObjectItemBuf;
import edu.rit.pj.Comm;
import edu.rit.pj.CommStatus;
import edu.rit.pj.LongForLoop;
import edu.rit.pj.LongSchedule;
import edu.rit.pj.ParallelRegion;
import edu.rit.pj.ParallelSection;
import edu.rit.pj.ParallelTeam;
import edu.rit.pj.reduction.LongOp;
import edu.rit.pj.reduction.SharedLong;
import edu.rit.util.LongRange;
import java.io.File;
public class PrimeCountFunctionHyb
   {
   // Sieve in one-million-number chunks.
   static final int CHUNK = 1000000;

   // World communicator.
   static Comm world;
   static int size;
   static int rank;

   // Command line arguments.
   static long x;
   static File primefile;
   static LongSchedule thrschedule;
```

We use a parallel thread team to calculate multiple sieves in parallel on the multiple processors of each hybrid parallel computer node.

```
   // Parallel team.
   static ParallelTeam team;
```

Each parallel team thread needs its own sieve object. However, we don't want to be continually creating sieve objects, each with a one-million-element `boolean` array. Instead, we create the per-thread sieve objects just once and reuse them.

```
   // Per-thread sieves.
   static Sieve[] sieves;
```

However, the threads can all share the same object to access the data set of odd primes.

```
   // List of 32-bit primes.
   static Prime32List primeList;
```

There also must be a shared reduction variable to hold the sum of the prime counts computed by the threads. To synchronize the threads, this variable is an instance of the multiple thread safe class SharedLong.

```
   // For counting primes.
   static SharedLong primeCount = new SharedLong (0);

   /**
    * Main program.
    */
   public static void main
      (String[] args)
      throws Exception
      {
      // Start timing.
      long t1 = System.currentTimeMillis();

      // World communicator.
      Comm.init (args);
      world = Comm.world();
      size = world.size();
      rank = world.rank();

      // Parse command line arguments.
      if (args.length < 2 || args.length > 3) usage();
      x = Long.parseLong (args[0]);
      if (x < 0) usage();
      primefile = new File (args[1]);
      thrschedule =
         args.length == 3 ?
            LongSchedule.parse (args[2]) :
            LongSchedule.fixed();
      // Set up parallel team and per-thread sieves.
      team = new ParallelTeam();
      sieves = new Sieve [team.getThreadCount()];
      for (int i = 0; i < sieves.length; ++ i)
         {
         sieves[i] = new Sieve (0, CHUNK);
         }

      // Set up list of 32-bit primes.
      primeList = new Prime32List (primefile);

      // In master process, run master section and worker section
      // in parallel.
      if (rank == 0)
         {
         new ParallelTeam(2).execute (new ParallelRegion()
            {
            public void run() throws Exception
               {
               execute (new ParallelSection()
                  {
                  public void run() throws Exception
                     {
                     masterSection();
                     }
                  },
               new ParallelSection()
                  {
                  public void run() throws Exception
                     {
                     workerSection();
                     }
                  });
               }
            });
         }

      // In worker process, run only worker section.
      else
         {
         workerSection();
         }
```

After all the calculations have concluded, the processes of the parallel program do a message-passing reduction to add up the prime counts, leaving the total in process 0. Process 0 then prints the answer.

```
      // Reduce prime counts into process 0.
      LongItemBuf buf = LongBuf.buffer (primeCount.longValue());
      world.reduce (0, buf, LongOp.SUM);

      // Stop timing.
      long t2 = System.currentTimeMillis();

      // Print the answer. (Add 1 because 2 is a prime.)
      if (rank == 0)
         {
         System.out.println ("pi("+x+") = "+(buf.item+1));
         }
      System.out.println ((t2-t1)+" msec "+rank);
      }

   /**
    * Perform the master section.
    */
   private static void masterSection()
      throws Exception
      {
      int worker;
      LongRange range;

      // Determine number of sieves to calculate.
      long ns = (x + CHUNK - 1) / CHUNK;
```

The master uses the `-Dpj.schedule` flag to partition the sieves among the parallel processes.

```
      // Set up a schedule object.
      LongSchedule schedule = LongSchedule.runtime();
      schedule.start (size, new LongRange (0, ns-1));

      // Send initial sieve range to each worker. If range is null,
      // no more work for that worker. Keep count of active workers.
      int activeWorkers = size;
      for (worker = 0; worker < size; ++ worker)
         {
         range = schedule.next (worker);
         world.send (worker, ObjectBuf.buffer (range));
         if (range == null) -- activeWorkers;
         }
      // Repeat until all workers have finished.
      while (activeWorkers > 0)
         {
         // Receive an empty message from any worker.
         CommStatus status =
            world.receive (null, IntegerBuf.emptyBuffer());
         worker = status.fromRank;

         // Send next chunk range to that specific worker. If null,
         // no more work.
         range = schedule.next (worker);
         world.send (worker, ObjectBuf.buffer (range));
         if (range == null) -- activeWorkers;
         }
      }

   /**
    * Perform the worker section.
    */
   private static void workerSection()
      throws Exception
      {
      // Process chunks from master.
      for (;;)
         {
         // Receive sieve range from master. If null, no more work.
         ObjectItemBuf<LongRange> rangeBuf = ObjectBuf.buffer();
         world.receive (0, rangeBuf);
         LongRange range = rangeBuf.item;
         if (range == null) break;
         final long lb = range.lb();
         final long ub = range.ub();
```

Each time the worker receives a block of work from the master, the worker reuses the same parallel thread team to do the calculations.

```
            // Calculate sieves in parallel threads.
            team.execute (new ParallelRegion()
               {
               public void run() throws Exception
                  {
                  execute (lb, ub, new LongForLoop()
                     {
                     // Per-thread variables plus extra padding.
```

```
Sieve thrSieve;
long thrPrimeCount;
long p0, p1, p2, p3, p4, p5, p6, p7;
long p8, p9, pa, pb, pc, pd, pe, pf;
```

The worker uses the final command-line argument to partition the sieves among the parallel threads.

```
// Use the thread-level loop schedule.
public LongSchedule schedule()
   {
   return thrschedule;
   }
```

Each thread reuses its own per-thread sieve object created back in the main program.

```
// Initialize per-thread variables.
public void start()
   {
   thrSieve = sieves[getThreadIndex()];
   thrPrimeCount = 0;
   }

// Calculate all sieves.
public void run (long first, long last)
   throws Exception
   {
   for (long lb = first; lb <= last; ++ lb)
      {
      // Get an iterator for the odd primes.
      LongIterator iter = primeList.iterator();

      // Calculate the sieve.
      thrSieve.lb (lb*CHUNK);
      thrSieve.initialize();
      thrSieve.sieveOut (iter);

      // Count primes <= x left in the sieve.
      iter = thrSieve.iterator();
      long p;
      while ((p = iter.next()) != 0 && p <= x)
         {
```

```
                        ++ thrPrimeCount;
                        }
                     }
                  }
```

The threads follow the SMP parallel reduction pattern. Each thread updates its own per-thread prime count. At the end of the parallel loop, each thread adds its per-thread prime count into the global reduction variable.

```
               // Reduce per-thread prime count into global
               // prime count.
               public void finish()
                  {
                  primeCount.addAndGet (thrPrimeCount);
                  }
               });
            }
         });

      // Report completion of sieve range to master.
      world.send (0, IntegerBuf.emptyBuffer());
      }
   };
}
```

Table 35.1 (at the end of the chapter) lists, and Figure 35.3 plots, the PrimeCountFunctionHyb program's performance on the "tardis" parallel computer. The program used a guided schedule for load balancing, both among the processes and among the threads. For the first set of runs, the program computed $\pi(2 \times 10^{10}) = 882{,}206{,}716$. For the second set of runs, the program computed $\pi(2 \times 10^{11}) = 8{,}007{,}105{,}059$.

We've now finished our study of parallel programming techniques on SMP, cluster, and hybrid parallel computers. In Part V, we'll put these techniques to use on three real-world, computation-intensive problems: spin relaxometry analysis of MRI images, protein sequence querying, and maximum parsimony phylogenetic tree construction.

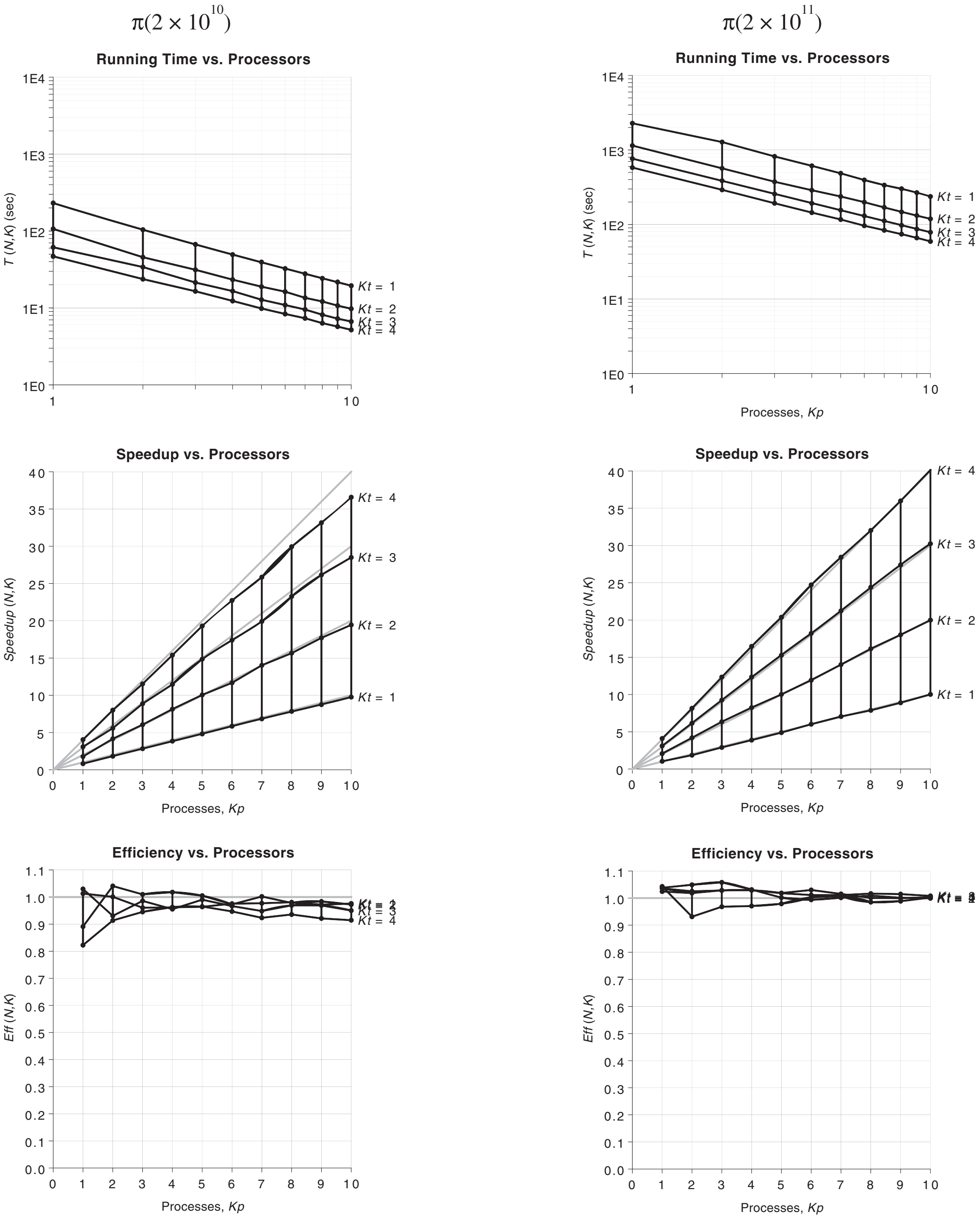


**Figure 35.3** PrimeCountFunctionSeq/Hyb running-time metrics

## 35.7 For Further Information

On the prime counting function:

- J. Derbyshire. *Prime Obsession: Bernhard Riemann and the Greatest Unsolved Problem in Mathematics.* Plume, 2003.
- E. Weisstein. “Prime Counting Function.” From *MathWorld*—A Wolfram Web Resource. http://mathworld.wolfram.com/PrimeCountingFunction.html
- E. Weisstein. “Prime Number Theorem.” From *MathWorld*—A Wolfram Web Resource. http://mathworld.wolfram.com/PrimeNumberTheorem.html

On the RSA public key cryptosystem:

- R. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2): 120–126, February 1978.
- N. Ferguson and B. Schneier. *Practical Cryptography.* Wiley Publishing, 2003.

**Table 35.1** PrimeCountFunctionSeq/Hyb running-time metrics

| $\pi(2\times10^{10})$ | | | | | | $\pi(2\times10^{11})$ | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| *Kp* | *Kt* | *T* | *Spdup* | *Eff* | *EDSF* | *Kp* | *Kt* | *T* | *Spdup* | *Eff* | *EDSF* |
| seq | seq | 189997 | | | | seq | seq | 2378300 | | | |
| 1 | 1 | 231003 | 0.822 | 0.822 | | 1 | 1 | 2280435 | 1.043 | 1.043 | |
| 1 | 2 | 106624 | 1.782 | 0.891 | -0.077 | 1 | 2 | 1145290 | 2.077 | 1.038 | 0.004 |
| 1 | 3 | 61506 | 3.089 | 1.030 | -0.101 | 1 | 3 | 765341 | 3.108 | 1.036 | 0.003 |
| 1 | 4 | 46900 | 4.051 | 1.013 | -0.063 | 1 | 4 | 580524 | 4.097 | 1.024 | 0.006 |
| 2 | 1 | 103984 | 1.827 | 0.914 | -0.100 | 2 | 1 | 1276921 | 1.863 | 0.931 | 0.120 |
| 2 | 2 | 45647 | 4.162 | 1.041 | -0.070 | 2 | 2 | 566810 | 4.196 | 1.049 | -0.002 |
| 2 | 3 | 34030 | 5.583 | 0.931 | -0.023 | 2 | 3 | 386697 | 6.150 | 1.025 | 0.003 |
| 2 | 4 | 23731 | 8.006 | 1.001 | -0.025 | 2 | 4 | 291897 | 8.148 | 1.018 | 0.003 |
| 3 | 1 | 67005 | 2.836 | 0.945 | -0.065 | 3 | 1 | 818917 | 2.904 | 0.968 | 0.039 |
| 3 | 2 | 31362 | 6.058 | 1.010 | -0.037 | 3 | 2 | 374672 | 6.348 | 1.058 | -0.003 |
| 3 | 3 | 21417 | 8.871 | 0.986 | -0.021 | 3 | 3 | 257072 | 9.251 | 1.028 | 0.002 |
| 3 | 4 | 16485 | 11.525 | 0.960 | -0.013 | 3 | 4 | 192681 | 12.343 | 1.029 | 0.001 |
| 4 | 1 | 49344 | 3.850 | 0.963 | -0.049 | 4 | 1 | 612509 | 3.883 | 0.971 | 0.025 |
| 4 | 2 | 23333 | 8.143 | 1.018 | -0.027 | 4 | 2 | 288260 | 8.251 | 1.031 | 0.002 |
| 4 | 3 | 16574 | 11.464 | 0.955 | -0.013 | 4 | 3 | 192543 | 12.352 | 1.029 | 0.001 |
| 4 | 4 | 12347 | 15.388 | 0.962 | -0.010 | 4 | 4 | 144477 | 16.461 | 1.029 | 0.001 |
| 5 | 1 | 39425 | 4.819 | 0.964 | -0.037 | 5 | 1 | 485892 | 4.895 | 0.979 | 0.016 |
| 5 | 2 | 18905 | 10.050 | 1.005 | -0.020 | 5 | 2 | 237433 | 10.017 | 1.002 | 0.005 |
| 5 | 3 | 12792 | 14.853 | 0.990 | -0.012 | 5 | 3 | 155750 | 15.270 | 1.018 | 0.002 |
| 5 | 4 | 9845 | 19.299 | 0.965 | -0.008 | 5 | 4 | 116748 | 20.371 | 1.019 | 0.001 |
| 6 | 1 | 32462 | 5.853 | 0.975 | -0.031 | 6 | 1 | 395173 | 6.018 | 1.003 | 0.008 |
| 6 | 2 | 16276 | 11.673 | 0.973 | -0.014 | 6 | 2 | 199424 | 11.926 | 0.994 | 0.004 |
| 6 | 3 | 10920 | 17.399 | 0.967 | -0.009 | 6 | 3 | 130662 | 18.202 | 1.011 | 0.002 |
| 6 | 4 | 8361 | 22.724 | 0.947 | -0.006 | 6 | 4 | 96220 | 24.717 | 1.030 | 0.001 |
| 7 | 1 | 27780 | 6.839 | 0.977 | -0.026 | 7 | 1 | 337203 | 7.053 | 1.008 | 0.006 |
| 7 | 2 | 13545 | 14.027 | 1.002 | -0.014 | 7 | 2 | 169645 | 14.019 | 1.001 | 0.003 |
| 7 | 3 | 9541 | 19.914 | 0.948 | -0.007 | 7 | 3 | 112019 | 21.231 | 1.011 | 0.002 |
| 7 | 4 | 7349 | 25.853 | 0.923 | -0.004 | 7 | 4 | 83658 | 28.429 | 1.015 | 0.001 |
| 8 | 1 | 24218 | 7.845 | 0.981 | -0.023 | 8 | 1 | 301769 | 7.881 | 0.985 | 0.008 |
| 8 | 2 | 12139 | 15.652 | 0.978 | -0.011 | 8 | 2 | 147287 | 16.147 | 1.009 | 0.002 |
| 8 | 3 | 8168 | 23.261 | 0.969 | -0.007 | 8 | 3 | 97480 | 24.398 | 1.017 | 0.001 |
| 8 | 4 | 6346 | 29.940 | 0.936 | -0.004 | 8 | 4 | 74263 | 32.025 | 1.001 | 0.001 |
| 9 | 1 | 21721 | 8.747 | 0.972 | -0.019 | 9 | 1 | 267280 | 8.898 | 0.989 | 0.007 |
| 9 | 2 | 10732 | 17.704 | 0.984 | -0.010 | 9 | 2 | 131985 | 18.019 | 1.001 | 0.002 |
| 9 | 3 | 7263 | 26.160 | 0.969 | -0.006 | 9 | 3 | 86812 | 27.396 | 1.015 | 0.001 |
| 9 | 4 | 5730 | 33.158 | 0.921 | -0.003 | 9 | 4 | 66096 | 35.983 | 1.000 | 0.001 |
| 10 | 1 | 19464 | 9.761 | 0.976 | -0.017 | 10 | 1 | 237365 | 10.020 | 1.002 | 0.005 |
| 10 | 2 | 9770 | 19.447 | 0.972 | -0.008 | 10 | 2 | 119011 | 19.984 | 0.999 | 0.002 |
| 10 | 3 | 6666 | 28.502 | 0.950 | -0.005 | 10 | 3 | 78649 | 30.239 | 1.008 | 0.001 |
| 10 | 4 | 5194 | 36.580 | 0.915 | -0.003 | 10 | 4 | 59264 | 40.131 | 1.003 | 0.001 |

*This page intentionally left blank*

# PART IV

# Exercises

*Exercises 1–8.* Here is a hybrid parallel version of Floyd's Algorithm for computing all shortest paths in an *n*-vertex graph whose distance matrix is *d*, an *n*-by-*n* matrix. Each element in *d* is a Java double-precision floating-point number. The middle and inner loops are executed by a parallel thread team. This version uses the scatter-gather pattern rather than the parallel input/output files pattern. The sequential version of Floyd's Algorithm is the same, without the communication operations or parallel team.

```
1  Scatter row slices of D from process 0 to each process
2  For i in 0 .. N-1
3      Broadcast row i of D to all processes
4      Parallel for r in this process's subrange of 0 .. N-1
5          For c in 0 .. N-1
6              D[r,c] = min (D[r,c], D[r,i] + D[i,c])
7  Gather row slices of D from each process into process 0
```

The distance matrix *d* is divided into equal-sized row slices among the processes. In each process, the slice is subdivided equally among the parallel team threads. Measurements show that executing the statement on line 6 takes 0.01 microseconds. Also, measurements show that sending a message from one process to another takes (400 + 0.8*B*) microseconds, where *B* is the number of bits of data in the message.

1. Give an expression for the running time $T_1$ in microseconds of the sequential version of Floyd's Algorithm as a function of *n*, the number of nodes. Ignore the loop overhead time.

2. Give an expression for the time $T_2$ in microseconds needed to send all the messages for the scatter operation on line 1 as a function of $n$, the number of nodes, $Kp$, the number of parallel processes, and $Kt$, the number of parallel threads per process (whichever variables are necessary).
3. Give an expression for the time $T_3$ in microseconds needed to send all the messages for the broadcast operation on line 3 as a function of $n$, the number of nodes, $Kp$, the number of parallel processes, and $Kt$, the number of parallel threads per process (whichever variables are necessary).
4. Give an expression for the running time $T_4$ in microseconds of the hybrid parallel version of Floyd's Algorithm as a function of $n$, the number of nodes, $Kp$, the number of parallel processes, and $Kt$, the number of parallel threads per process (whichever variables are necessary). Ignore the loop overhead time.
5. On a hybrid parallel computer where each node is a dual-core PC, how many parallel processes should be used to obtain the smallest running time for a 2,000-vertex graph? (Note that the number of parallel processes must be an integer.)
6. On a hybrid parallel computer where each node is a dual-core PC, what is the largest speedup that can be obtained for a 2,000-vertex graph?
7. On a hybrid parallel computer where each node is an 8-CPU SMP server, how many parallel processes should be used to obtain the smallest running time for a 2,000-vertex graph? (Note that the number of parallel processes must be an integer.)
8. On a hybrid parallel computer where each node is an 8-CPU SMP server, what is the largest speedup that can be obtained for a 2,000-vertex graph?

*Exercises 9–15.* Given an integer $i > 0$, consider the following procedure:

```
$x \leftarrow i$
While $x > 1$:
    If $x$ is even:
        $x \leftarrow x/2$
    Else:
        $x \leftarrow 3x+1$
```

The **Collatz Conjecture**, proposed by Lothar Collatz in 1937, states that for every $i > 0$, the preceding procedure terminates; that is, $x$ eventually becomes 1. Although mathematicians believe the Collatz Conjecture is true, no one has been able to prove it yet.

9. Write a sequential program to investigate whether the Collatz Conjecture is true for all values of $i$ from 1 through $N$, where $N$ is a command line argument. Use type `long` so $N$ can be as large as $2^{63}–1$. The program also has a command-line argument *MaxIter* (type `long`). In the preceding procedure,

for a certain value of *i*, if *x* reaches 1 before the number of while loop iterations reaches *MaxIter*, then the Collatz Conjecture is true for *i*. If the number of while loop iterations reaches *MaxIter* before *x* reaches 1, then the Collatz Conjecture *may* be false for *i*. (The Collatz Conjecture is not *definitely* false for *i* because *x* might reach 1 with further iterations, but the program has to stop somewhere.) The program prints the values of *i* for which the Collatz Conjecture may be false.

10. Write a hybrid parallel program to investigate the Collatz Conjecture. The parallel program has the same command-line arguments and the same output as the sequential program. The parallel program uses one-level scheduling.

11. Write a hybrid parallel program to investigate the Collatz Conjecture. The parallel program has the same command-line arguments and the same output as the sequential program. The parallel program uses two-level scheduling; the thread-level schedule is specified as an additional command-line argument.

12. Measure the two parallel programs' running times as a function of *N*, *Kp* (number of processes), and *Kt* (number of threads per process), using a fixed schedule for the processes and the threads. Calculate the program's running-time metrics.

13. Measure the two parallel programs' running times as a function of *N*, *Kp* (number of processes), and *Kt* (number of threads per process), using a dynamic schedule of an appropriate size for the processes and the threads. Calculate the program's running-time metrics.

14. Measure the two parallel programs' running times as a function of *N*, *Kp* (number of processes), and *Kt* (number of threads per process), using a guided schedule for the processes and the threads. Calculate the program's running-time metrics.

15. Based on your data, which schedule gives the best performance? Explain why.

*Exercises 16–20.* A **three-dimensional random walk** is defined as follows. A particle is initially positioned at (0, 0, 0) in the X-Y-Z coordinate space. The particle does a sequence of *N* steps. At each step, the particle chooses one of the six directions left, right, ahead, back, up, or down at random, then moves one unit in that direction. Specifically, if the particle is at (*x*, *y*, *z*):

With probability 1/6 the particle moves left to (*x*–1, *y*, *z*).
With probability 1/6 the particle moves right to (*x*+1, *y*, *z*).
With probability 1/6 the particle moves back to (*x*, *y*–1, *z*).
With probability 1/6 the particle moves ahead to (*x*, *y*+1, *z*).
With probability 1/6 the particle moves down to (*x*, *y*, *z*–1).
With probability 1/6 the particle moves up to (*x*, *y*, *z*+1).

16. Write a sequential program to calculate the particle's final position. The program's command-line arguments are the random seed and the number of steps *N*. The program prints the particle's final position (*x*, *y*, *z*) as well as the particle's final distance from the origin.

17. Describe the sequential dependencies, if any, in the program. Is it possible to parallelize the program?

18. If possible, write a hybrid parallel program to calculate the particle's final position. The parallel program has the same command-line arguments and the same output as the sequential program. Measure the parallel program's running times as a function of *N*, *Kp* (number of processes), and *Kt* (number of threads per process), calculate the program's running-time metrics, and improve the program's design, if necessary.

19. What is the particle's expected final distance from the origin as a function of the number of steps *N*?

20. Run your program for a large number of steps and a variety of different random seeds. Do the particle's computed final distances from the origin agree with the expected final distance?

*Exercises 21–25.* In 1955, the RAND Corporation produced a book titled *A Million Random Digits with 100,000 Normal Deviates*. The first part of the book was just as the title says, one million random digits (0–9), printed in groups of five for readability. The digits were generated by a hardware electronic simulation of a roulette wheel using an environmental noise source for randomness.

21. Write a sequential program to produce a file of random digits. The program must take three command-line arguments: a random seed (type `long`); the number of digits to produce, *N* (type `long`); and the output file name. The program must initialize a pseudorandom number generator with the given seed; generate *N* random numbers each uniformly distributed between 0 and 9 inclusive; and write the numbers into a plain text file with the given output file name. There are to be no spaces or newlines in the output file, just the numbers.

22. Design a hybrid parallel version of this program. The program must be designed so that if run with *Kp* parallel processes and *Kt* parallel threads per process, the speedup will be as close to *Kp*·*Kt* as possible. The program must be designed so that for a given seed and *N*, the output file produced is always the same as the sequential version, regardless of what *Kp* and *Kt* are. The program must be designed so that only one process writes the output file. Describe your design in detail. Be especially clear in your description of any message passing operations needed: describe what the message passing operations are; describe which process and which data structure the data comes from; and describe which process and which data structure the data goes into. Also, be especially clear in your description of how the program generates the random numbers.

23. Design another hybrid parallel version of this program with the same requirements as Exercise 22, except that each process writes its own section of the output file. (Assume all processors can access the same output file.) Describe your design in detail. Be especially clear in your description of any message passing operations needed: describe what the message passing operations are; describe which process and which data structure the data comes from; and describe which process and which data structure the data goes into. Also, be especially clear in your description of how the program generates the random numbers.

24. Without actually measuring the running times, which hybrid parallel version do you expect will have better performance? Explain why.

25. Implement the sequential and the two hybrid parallel versions using Parallel Java. Measure the running times as a function of *N*, *Kp*, and *Kt*, and calculate the running-time metrics. Do your measurements support your hypothesis from Exercise 24? If not, explain why not.

26. Write a hybrid parallel program to calculate $\pi_2(x)$, the number of **twin primes** less than or equal to *x*. The value of *x* is given as a command-line argument. If *p* is a prime and *p*+2 is a prime, then (*p*, *p*+2) are called twin primes. The first few twin primes are (3, 5), (5, 7), (11, 13), (17, 19), (29, 31), and (41, 43).
*Caution:* If *p* is a member of more than one pair of twin primes—like *p*=5—then *p* is counted only once when computing $\pi_2(x)$.
*Caution:* If your program uses sieving in multiple chunks like the PrimeCountFunctionHyb program, keep in mind that the two primes of a pair may be split across two chunks; furthermore, these chunks may be calculated in different processors.

27. Write a hybrid parallel program to calculate $\pi_i(x)$, the number of primes less than or equal to *x* such that *p* and *p*+*i* are both prime. The values of *x* and *i* are given as command-line arguments. For *i*=2, (*p*, *p*+2) are called twin primes. For *i*=4, (*p*, *p*+4) are called **cousin primes**. For *i*=6, (*p*, *p*+6) are called **sexy primes** (from the Latin word for "six," of course!).

*Exercises 28–29.* Alice, the computational astrophysicist, ran a gravitational *N*-body program to compute the motion of 10,000 stars in a star cluster. The program generated an output file consisting of 1,000,000 snapshots at equal time intervals. Each snapshot consists of the position and velocity of each star in the star cluster. Each position and velocity consists of the X, Y, and Z coordinates. Each coordinate is stored as a 4-byte `float`.

28. What is the output file's length in bytes?

29. Alice wants to analyze the program's results to detect whether any stars escaped from the cluster. A star escapes from the cluster if the magnitude of the star's velocity ever exceeds a specified threshold (the **escape velocity** of the cluster). Every star that escaped is to be reported, along with the time at

which it exceeded the escape velocity. Alice wants to use a parallel program to speed up the analysis. Which parallel data-set querying strategy should she use? Justify your answer.

*Exercises 30–31.* Barbara, the computational biologist, has a group of 2,000 protein sequences. Each sequence is a string of characters; each character stands for one amino acid. The average length of the protein sequences is 500 characters. Each character is stored in one byte.

30. What is the data set's size in bytes?
31. Barbara wants to discover which proteins are the most similar to each other. To compute the similarity of two proteins, run a "local alignment" algorithm on the two proteins, which yields a score (a single number). The local alignment algorithm's running time is O($m{\cdot}n$), where $m$ and $n$ are the lengths of the two sequences being compared. Higher scores indicate more similar proteins. All pairs of proteins with a similarity score above a specified threshold are to be reported. Barbara wants to use a parallel program to speed up the analysis. Which parallel data-set querying strategy should she use? Justify your answer.

*Exercises 32–33.* Up until now, we've assumed that a hybrid parallel computer is **homogeneous**: each backend node has the same number of CPUs, and all the CPUs on all the nodes run at the same speed. Let's change the first part of this assumption: different backend nodes have different numbers of CPUs; however, all the CPUs on all the nodes still run at the same speed. Keep in mind that when writing a hybrid parallel program, you don't know how many CPUs there will be on each node when you run the program.

32. Under the new assumptions, do you expect the AES key search hybrid parallel program in Chapter 32 still to experience good speedups as the number of processors scales up? If so, explain why. If not, explain why not, and describe how you would change the program's design to get good speedups.
33. Under the new assumptions, do you expect the Mandelbrot Set hybrid parallel program in Chapter 33 still to experience good speedups as the number of processors scales up? If so, explain why. If not, explain why not, and describe how you would change the program's design to get good speedups.

# PART V Applications

*This page intentionally left blank*

# CHAPTER 36

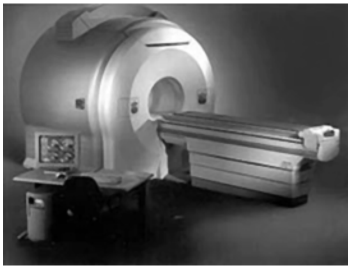
# MRI Spin Relaxometry

in which we learn a bit about how an MRI scanner works; we state the problem of analyzing MRI signals to determine spin relaxation rates; we derive an algorithm to perform the analysis; we implement the algorithm as a cluster parallel program; and we speculate how the results could help diagnose disease

# CHAPTER 36 MRI Spin Relaxometry

## 36.1 MRI Scanning

A **magnetic resonance image (MRI) scanner**, like the one in Figure 36.1, can be used to take pictures of a person's insides. Although the pictures are similar to those obtained using X-rays, an MRI scanner does not use high-energy radiation and is therefore potentially less harmful than an X-ray machine. In addition, MR images can be analyzed to yield information about the chemical composition of the tissues being scanned. The analysis requires lengthy computations; however, the computations can be done in a massively parallel fashion. In this chapter, we are going to design a parallel program to carry out such an analysis. Before we can do that, we need some background about the physics and mathematics of MRI scanning.



Courtesy of NASA. http://rst.gsfc.nasa.gov/Intro/mri.jpg

**Figure 36.1** An MRI scanner

An MRI scanner uses the **spins** of atomic nuclei, primarily the nuclei of hydrogen atoms, which are abundant in the body. Spin is a quantum mechanical property that makes the atomic nucleus behave like a tiny magnet. If a bunch of atoms are placed in a strong magnetic field, the atoms' spins tend to align themselves with the field. The preferred spin alignment is in the same direction as the field, like a compass needle pointing north; in this case, the atom is in a low-energy state. However, the spin can also align in the direction opposite to the field, like a compass needle pointing south; in this case, the atom is in a high-energy state.

Consider a large number of atoms in a region of the body under the influence of an external magnetic field. Some of the spins will be aligned with the field, others against the field. Assign a number called the **magnetization** to each atom. The magnetization is positive if the spin is aligned with the field; it is negative if the spin is aligned against the field. Now add up the magnetizations of all the atoms in the region to get the **net magnetization**. If most of the spins are aligned against the field, the net magnetization will be negative; if most of the spins are aligned with the field, the net magnetization will be positive; if equal numbers are aligned with and against the field, the net magnetization will be zero.

Suppose we impose such an external magnetic field on a region of the body. Most of the atoms will go into the low-energy state, with spins aligned with the field, resulting in a positive net magnetization. Now suppose we send a pulse of energy through the region—energy of just the right frequency to flip the atoms' spins to the high-energy state. The frequency required depends on the type of atom and the strength of the external magnetic field. For hydrogen atoms in a 1-Tesla field, the frequency is 42.6 megahertz—a radio frequency (RF), far lower than X-ray frequencies. The region's net magnetization is now negative. However, as time passes, the atoms tend to flip back to the low-energy state. The net magnetization increases from a negative value toward zero. As more time passes, more atoms flip to the low-energy state, and the net magnetization becomes positive. Eventually, the net magnetization returns to its original value. A plot of net magnetization versus time looks like Figure 36.2.

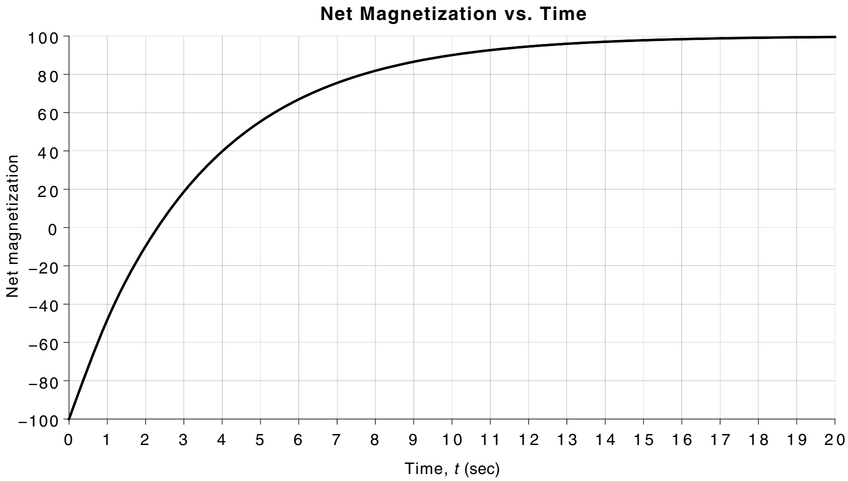


**Figure 36.2** Net magnetization versus time for inversion recovery sequence

An MRI scanner uses massive superconducting magnets to impose a strong external magnetic field along the axis of the scanner, through the cavity in the center where the patient lies (see Figure 36.1). The MRI scanner then emits a pulse of RF energy. The RF pulse flips, or "inverts," the spins of the atoms in the patient. As the spins flip back, or "recover" to their original states, the MRI scanner measures the net magnetization within a thin **slice** perpendicular to the axis (Figure 36.3). The slice's position is selected to pass through a section of the brain, or knee, or whatever part of the anatomy the doctor wants to examine. This measurement procedure is called the **inversion recovery sequence**. (MRI scanners can also perform other measurement sequences that we will not discuss.)

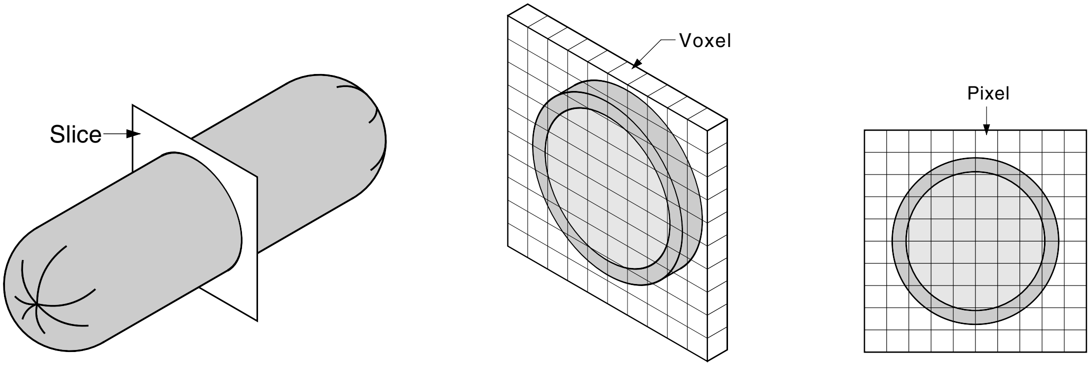


**Figure 36.3** MRI scan of a salami, illustrating a slice, voxels, and pixels

Rather than just measure the net magnetization signal, or **spin signal**, for the entire slice, the MRI scanner subdivides the slice into a rectangular grid. Each grid element, or **voxel**, corresponds to a small volume inside the patient. The MRI scanner measures the spin signal separately for each voxel and records the data as a two-dimensional image. Each pixel in the image corresponds to a voxel in the slice.

The MRI scanner takes a series of snapshots of the slice's spin signals. Soon after the RF pulse that inverts the spins, when the net magnetizations are mostly negative, the MRI scanner takes the first snapshot. The MRI scanner continues taking snapshots at certain time intervals as the spins relax back to their original states and the net magnetizations become positive again. For example, Figure 36.4 depicts a series of spin signal measurements of a slice through someone's brain. Each snapshot is shown as a 64 × 64-pixel image. Each pixel's gray level corresponds to the measured spin signal of a voxel at that time, with negative values being dark and positive values being light.

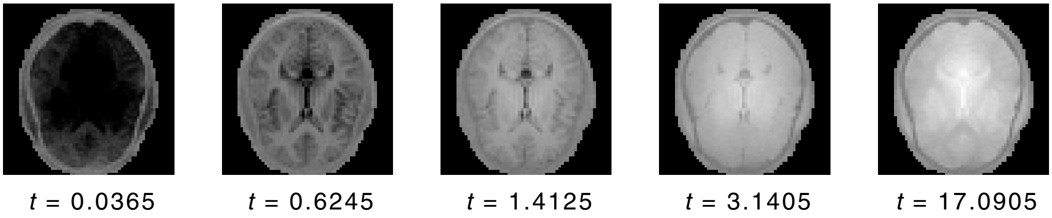


**Figure 36.4** Sequence of spin signal MR images

To extract the spin signal as a function of time for a particular voxel, we have to take the corresponding pixel's measurement from each of the images (Figure 36.5). This gives a series of spin signal values taken at a series of time values. Figure 36.6 shows the spin signal for the pixel in row 32, column 46 (pixel number 2,094) of the example data set. This data set consists of 64 images; thus, each pixel's spin signal series is 64 elements long, as is the time series.

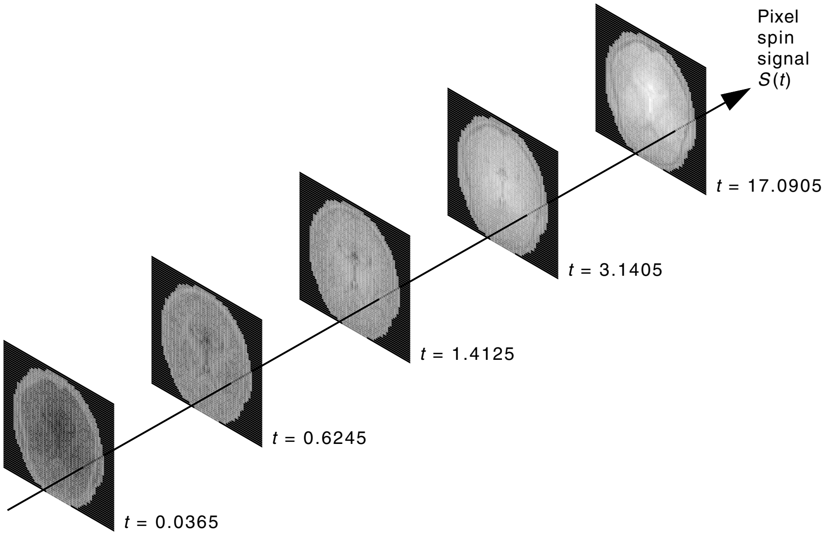


**Figure 36.5** Extracting the spin signal for one pixel

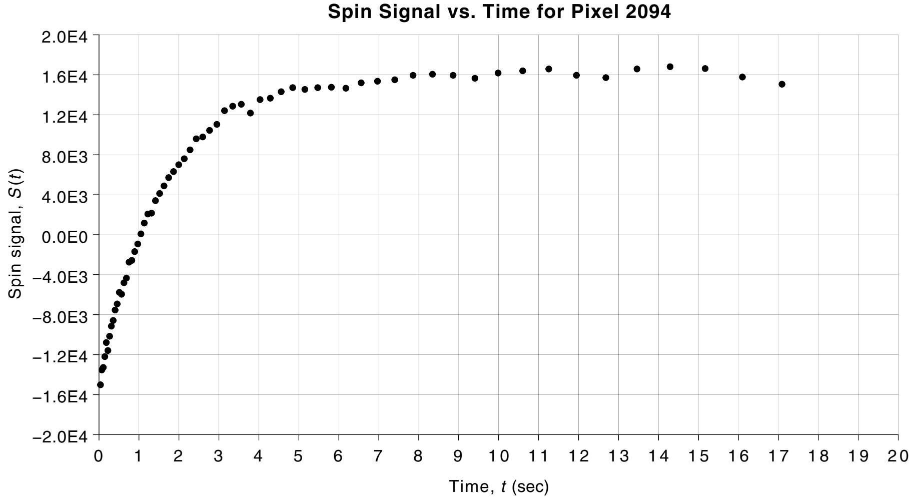


**Figure 36.6** Spin signal for pixel 2,094

The series of images we have examined is for *one* slice through the patient. In a typical MRI scan, however, the scanner takes images of *many* slices through the patient, each slice at a slightly different location along the axis (Figure 36.7). Stacking up the slices yields a three-dimensional image of the patient's insides. If we have a 64 × 64 × 64-voxel MR image, and each voxel has a spin signal series with 64 elements, there are nearly 17 million spin signal values to analyze—and this is a rather small MRI scan. An MRI scanner typically takes image snapshots of 256 × 256 pixels. Perhaps you're starting to get an idea of the amount of computation it takes to analyze an MR image.

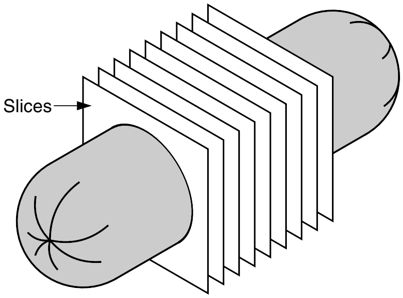


**Figure 36.7** Three-dimensional MRI scan

## 36.2 Spin Relaxometry Analysis

The inversion recovery sequence spin signal for a voxel, $S(t)$, obeys this formula,

$$S(t) = \rho\ [1 - 2\exp(-R_1\ t)] \tag{36.1}$$

where $\rho$ is the **spin density** and $R_1$ is the **spin-lattice relaxation rate**. The spin signal relaxes from $-\rho$ at $t$=0 to + $\rho$ at $t$=∞. $R_1$ determines the rate at which the spin signal relaxes. If $R_1$ is small, $S$ relaxes slowly and takes a long time to reach its asymptotic value. If $R_1$ is large, $S$ relaxes quickly. The **spin-lattice relaxation time** $T_1$ is $1/R_1$; if $R_1$ is large, $T_1$ is small, and vice versa. $T_1$ is measured in seconds (sec) or milliseconds (msec). $R_1$ is measured in sec$^{-1}$.

The spin density and spin-lattice relaxation rate depend on the type of tissue in the voxel. $\rho$ depends on the abundance of hydrogen atoms in the tissue. $R_1$ depends on the kinds of molecules in the tissue; the spins of hydrogen atoms in different molecules relax at different rates. Here are the characteristic spin-lattice relaxation times and rates of six types of tissues found in the brain (from the paper "A multispectral analysis of brain tissues" by Fletcher *et al.*):

| Tissue | $T_1$ (sec) | $R_1$ (sec$^{-1}$) |
|---|---|---|
| Cerebrospinal fluid | 0.80 – 20.0 | 0.05 – 1.25 |
| White matter | 0.76 – 1.08 | 0.93 – 1.32 |
| Gray matter | 1.09 – 2.15 | 0.47 – 0.92 |
| Meninges | 0.50 – 2.20 | 0.45 – 2.00 |
| Muscle | 0.95 – 1.82 | 0.55 – 1.05 |
| Adipose | 0.20 – 0.75 | 1.33 – 5.00 |

At this point, we change the notation slightly from what MRI physicists use. $R_1$ has a subscript "1" because there is another relaxation rate, the spin-spin relaxation rate, symbolized as $R_2$. The spin-spin relaxation rate doesn't appear in the inversion recovery sequence. To avoid a confusing multiplicity of subscripts in the formulas we are about to study, we will use just the symbol $R$ for the spin-lattice relaxation rate.

Each voxel typically encompasses more than one kind of tissue. The measured spin signal is then the superposition, or sum, of several individual spin signals,

$$S(t) = \sum_{j=1}^{N} \rho_j [1 - 2\exp(-R_j t)] \tag{36.2}$$

where $N$ is the number of tissues, $\rho_j$ is the spin density for the $j$-th tissue, and $R_j$ is the spin-lattice relaxation rate for the $j$-th tissue.

**Spin relaxometry** is the process of analyzing the spin signal for a voxel, like Figure 36.6, to determine the spin densities and spin-lattice relaxation rates of the tissues in the voxel, $\rho_j$ and $R_j$. The relaxation rates in particular give clues about the tissues' molecular composition. In healthy tissue, normal $R$ values are observed. In diseased tissue, abnormal chemicals may be present and abnormal $R$ values may be observed. Thus, a doctor can use a spin relaxometry analysis of an MRI scan to detect disease.

We are facing a curve-fitting problem. We need to find the number of tissues $N$ and the parameters $\rho_j$ and $R_j$ for $1 \le j \le N$ so that the theoretical model for the signal, the right side of Equation 36.2, best matches the actual measured signal $S(t)$ on the left side. We will use the usual least-squares criterion and find the parameters that minimize the sum of the squares of the differences between the measured values and the model values.

Because the model is a *nonlinear* function of the $R_j$ parameters, we must use a **nonlinear least-squares** curve-fitting procedure. However, the nonlinear least-squares algorithm requires an initial estimate of the solution; we must know how many tissues there are ($N$) and roughly what their parameters are ($\rho_j$ and $R_j$). The algorithm then adjusts the parameters to yield the best curve fit. If we start the algorithm with the wrong number of tissues, or with initial parameter estimates that are too far off, the algorithm cannot find a good solution.

We can use a **linear least-squares** curve-fitting procedure to solve the problem partway and come up with an initial estimate to feed into the nonlinear least-squares algorithm. Define the **model function** $f$ to be the following:

$$f(R,t) = 1 - 2\exp(-R\ t) \tag{36.3}$$

Then, the theoretical spin signal for the $i$-th time sample $t_i$ is the following:

$$\rho_1\ f(R_1, t_i) + \rho_2\ f(R_2, t_i) + \ldots + \rho_N\ f(R_N,\ t_i) \tag{36.4}$$

Note that (36.4) is a linear function of the $f$ values. Let there be $M$ time samples $t_1$ through $t_M$ and $M$ spin signal samples $S_1$ through $S_M$, where $S_i = S(t_i)$. Repeating (36.4) for the $M$ time samples and equating these to the $M$ signal samples gives the following:

$$\begin{gathered}
\rho_1\ f(R_1, t_1) + \rho_2\ f(R_2, t_1) + \ldots + \rho_N\ f(R_N,\ t_1) = S_1 \\
\rho_1\ f(R_1, t_2) + \rho_2\ f(R_2, t_2) + \ldots + \rho_N\ f(R_N,\ t_2) = S_2 \\
\vdots \\
\rho_1\ f(R_1, t_M) + \rho_2\ f(R_2, t_M) + \ldots + \rho_N\ f(R_N,\ t_M) = S_M
\end{gathered} \tag{36.5}$$

Or, expressed more compactly in matrix notation,

$$\mathbf{F} \cdot \rho = \mathbf{S} \tag{36.6}$$

where **F**, known as the **design matrix** of the linear least-squares problem, is an $M \times N$-element matrix with $F_{ij} = f(R_j, t_i)$, $\rho$ is an $N$-element vector of spin density values, and S is an $M$-element vector of measured signal values.

Suppose we pick a series of $N$ relaxation rates $R_j$, $1 \le j \le N$, that span the range of relaxation rates we expect to find in the tissues. For example, we might pick 200 relaxation rates from $R_1 = 0.01$ to $R_{200} = 10.0$. We also know the $M$ time values $t_i$, $1 \le i \le M$. From this information, we can calculate **F**. We then can solve the linear system of equations (36.6) for the unknown $\rho$, given the known **F** and **S**. One further property of our model turns out to be crucial for a successful solution. The spin densities must all be *nonnegative* ($\rho_j \ge 0$). We therefore use a **nonnegative linear least-squares** algorithm to find the spin densities that give the best fit between the model and the measured spin signal for the chosen relaxation rates. Class edu.rit.numeric.NonNegativeLeastSquares in the Parallel Java Library implements the algorithm. It is a translation into Java of the public domain Fortran subroutine `NNLS` by Charles Lawson and Richard Hanson.

Figure 36.8 shows the nonnegative linear least-squares solution of (36.6) for pixel 2,094 of the example data set using 200 relaxation rates logarithmically spaced from 0.01 to 10.0. Each dot plots $\rho_j$ on the vertical axis versus $R_j$ on the horizontal axis. All but a few of the spin densities are zero. Specifically, of the 200 potential tissues (relaxation rates), only the following are actually present (have nonzero spin densities):

| $\rho_j$ | $R_j$ |
|---:|---:|
| 796 | 0.285 |
| 6332 | 0.631 |
| 8291 | 0.653 |
| 770 | 10.000 |

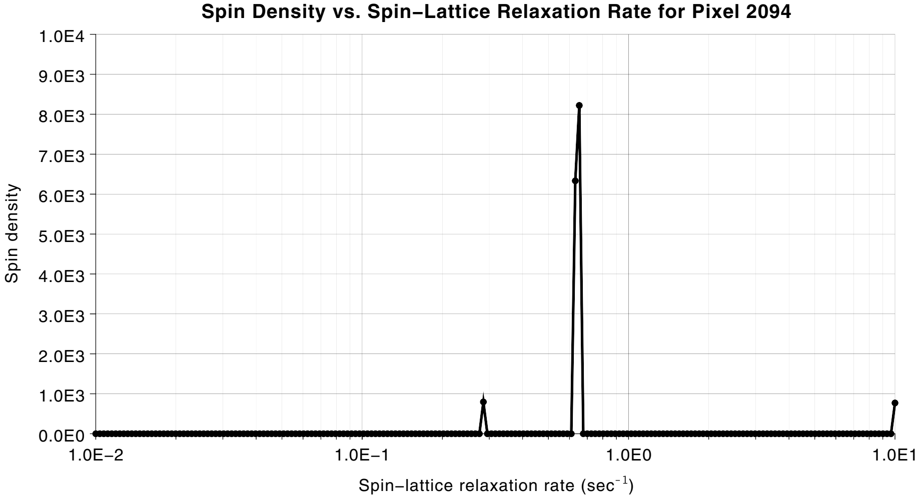


**Figure 36.8** Nonnegative linear least-squares solution for pixel 2,094

There are three peaks in the linear least-squares solution, indicating that pixel 2,094 has three tissues. (A peak occurs at index *j* if $\rho_j > \rho_{j\text{-}1}$ and $\rho_j > \rho_{j+1}$.) The first tissue has a relaxation rate somewhere around 0.285. The second tissue has a relaxation rate somewhere around 0.631–0.653. The third tissue has a relaxation rate of 10.0 or greater.

The linear least-squares solution gives the model (Equation 36.2) for the nonlinear least-squares curve fit. The number of tissues *N* is number of peaks. The peaks' densities and relaxation rates are the initial estimates for the parameters $\rho_j$ and $R_j$. For pixel 2,094, the initial estimates are the following:

| $\rho_j$ | $R_j$ |
|---|---|
| 796 | 0.285 |
| 8291 | 0.653 |
| 770 | 10.000 |

To do the nonlinear least-squares curve fit, we use the **Levenberg-Marquardt algorithm**. This algorithm finds the minimum of the sum of the squares of a series of functions. For the spin relaxometry analysis, the functions are the differences between the theoretical spin signal and the actual spin signal for the *M* time samples, namely the following:

$$\begin{aligned} f_1 &= \left( \sum_{j=1}^{N} \rho_j \ [1 - 2\exp(-R_j t_1)] \right) - S_1 \\ f_2 &= \left( \sum_{j=1}^{N} \rho_j \ [1 - 2\exp(-R_j t_2)] \right) - S_2 \\ &\vdots \\ f_M &= \left( \sum_{j=1}^{N} \rho_j \ [1 - 2\exp(-R_j t_M)] \right) - S_M \end{aligned} \tag{36.7}$$

Starting from the initial estimates for the parameters $\rho_j$ and $R_j$, the algorithm adjusts the parameter values to minimize the sum of the squares of the functions (36.7). That is, the algorithm does a least-squares curve fit of the model spin signal to the measured spin signal. The algorithm does not require the model to be a linear function of the parameters, and, in fact, the algorithm is designed especially for nonlinear functions.

To do its job, the Levenberg-Marquardt algorithm needs to calculate the partial derivatives of the model functions (36.7) with respect to the parameters $\rho_j$ and $R_j$. The partial derivatives are the following:

$$\frac{\partial f_i}{\partial \rho_j} = 1 - 2\exp(-R_j t_i) \tag{36.8}$$

$$\frac{\partial f_i}{\partial R_j} = 2\,\rho_j t_i \exp(-R_j t_i) \tag{36.9}$$

For a given set of parameter values, the Levenberg-Marquardt algorithm uses the partial derivatives to calculate the **gradient**, or slope, of the sum of the squares of the functions (36.7). Adjusting the parameters in the direction of decreasing gradient makes the sum-of-squares smaller. By adjusting the parameters iteratively in a series of small steps, the algorithm eventually finds a spot in parameter space where the gradient is zero and any changes in the parameters cause the sum-of-squares to increase—that is, a minimum of the sum-of-squares. When the algorithm finishes, the final parameter values give the nonlinear least-squares curve fit.

More precisely, the Levenberg-Marquardt algorithm finds the spot in parameter space *closest to the initial parameter estimates* where the gradient is zero; that is, a "local minimum." To yield good results, the algorithm must start with initial parameter estimates close to the final solution. The purpose of the *linear* least-squares algorithm is to find this nearby starting point for the *nonlinear* least-squares algorithm.

Details of how the Levenberg-Marquardt algorithm operates are beyond the scope of this book. Class edu.rit.numeric.NonLinearLeastSquares in the Parallel Java Library implements the algorithm. It is a translation into Java of the public domain Fortran subroutine `LMDER`, part of the MINPACK Library, by Jorge Moré, Burt Garbow, and Ken Hillstrom.

For pixel 2,094 in the example data set, here are the final parameter values after the nonlinear least-squares curve fit:

| $\rho_j$ | $R_j$ |
|---:|---:|
| 1049 | 0.262 |
| 14437 | 0.659 |
| 697 | 17.289 |

Figure 36.9 plots the fitted curve for pixel 2,094 with the preceding parameters, along with the original measured spin signal data points.

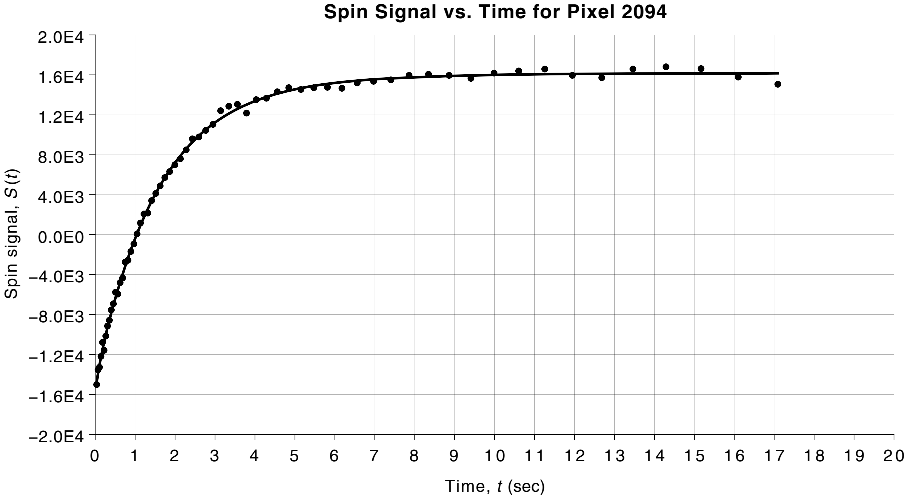


**Figure 36.9** Nonlinear least-squares solution for pixel 2,094

The symbol $\chi^2$ denotes the sum over all the data points of the squares of the differences between the fitted curve and the measured signal. The smaller the $\chi^2$, the closer the fit. After the linear least-squares curve fit for pixel 2,094, $\chi^2$ was $9.19\times10^6$. After the nonlinear least-squares curve fit, $\chi^2$ was $9.06\times10^6$. By adjusting the tissue parameter values found by the linear algorithm, the nonlinear algorithm came up with a better fit.

## 36.3 Sequential Spin Relaxometry Program

Now that we have a technique for analyzing one pixel, we can write a program to do a spin relaxometry analysis of an entire MRI scan. We'll begin with a sequential version. Figure 36.10 shows the overall analysis workflow. The first step is to convert the raw MR image data produced by the MRI scanner to a standard format stored in a group of spin signal data set files. Each file holds the time series $t_i$ and the pixels' spin signal series $S_i$ for one slice of the MRI scan. CreateSignalDataSet is a program that does this conversion for the example MR image we've been using. Because this program doesn't do any lengthy processing—it just shuffles the data around—it's not worth the effort to make it a parallel program.

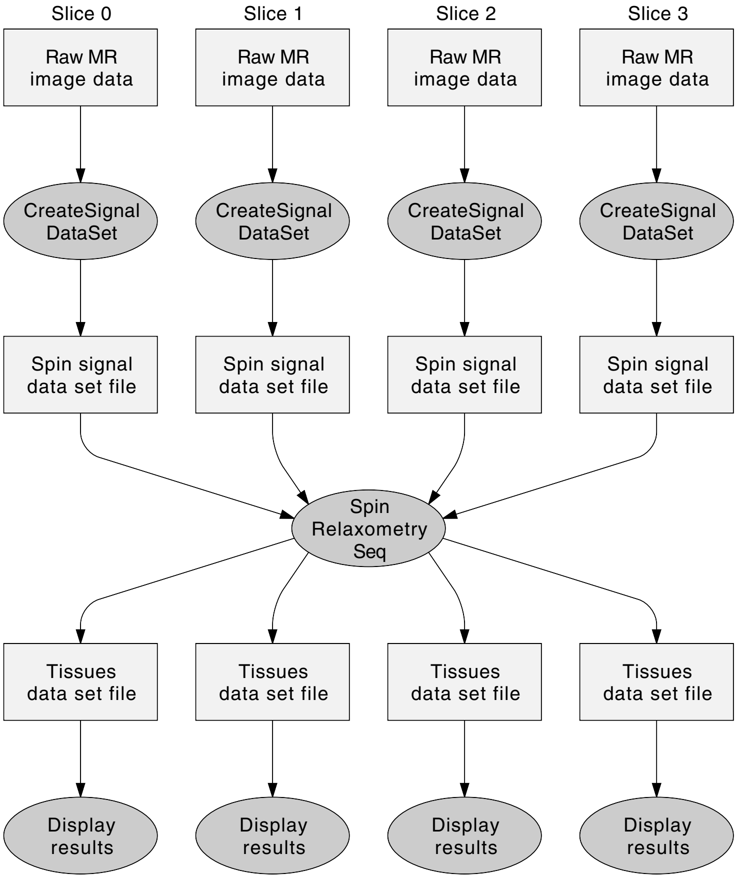


**Figure 36.10** MRI spin relaxometry analysis workflow

The second step is to do the spin relaxometry analysis for every pixel of every slice of the MRI scan. This is where all the time-consuming calculations happen, and this is the part we will parallelize. Class SpinRelaxometrySeq is the sequential version of the analysis program. For each spin signal data set file, the program writes a tissues data set file consisting of the number of tissues and the $\rho_j$ and $R_j$ values for each pixel.

Once the analysis is complete, other (non-parallel) programs postprocess and display the results stored in the tissues data set files. We will look at examples of these postprocessing programs later.

Figure 36.11 shows the classes from which the sequential spin relaxometry analysis program is built and their "uses" relationships. (A → B means class A uses class B.) Unless otherwise noted, these classes are in package edu.rit.mri. We'll describe what every class does, but study the code for only a few key classes.

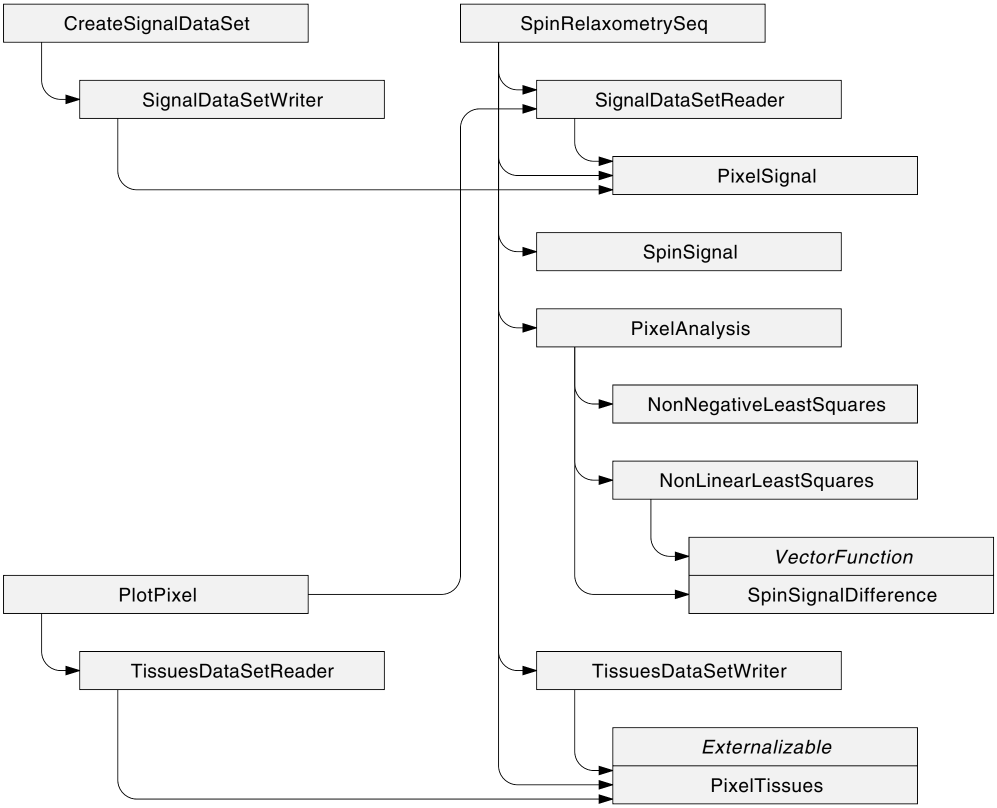


**Figure 36.11** Spin relaxometry analysis programs' class relationships

- Classes SignalDataSetReader and SignalDataSetWriter provide objects that read and write, respectively, a spin signal data set file for one slice of an MRI scan (one MR image). The file contains the series of time values at which the MRI scanner took its measurements as well as the series of spin signal values for each pixel in the image. The file also has an index that gives the offset of each pixel's spin signal series. Given a pixel index, the SignalDataSetReader object reads the offset at that index, and then seeks directly to that offset to read the spin signal. This allows the program to quickly read an arbitrary pixel's spin signal.

- Class PixelSignal provides an object containing the series of spin signal values for one pixel, along with the pixel index and an index designating the MR image. Class SignalDataSetReader reads the spin signal data set file and returns PixelSignal objects. Class SignalDataSetWriter takes PixelSignal objects and writes them to the file.
- As already mentioned, class CreateSignalDataSet is a main program that takes raw MR image data and writes it to a file using a SignalDataSetWriter, so the other programs can read the data using a SignalDataSetReader.
- Classes TissuesDataSetReader and TissuesDataSetWriter provide objects that read and write, respectively, a tissues data set file for one slice of an MRI scan (one MR image). For each pixel in the image, the file contains the number of tissues and the spin density and spin-lattice relaxation rate for each tissue. Like the spin signal data set file, the tissues data set file also has an index that gives the offset of each pixel's tissues data.
- Class PixelTissues provides an object containing the tissue parameters for one pixel, along with the pixel index and an index designating the MR image. Class TissuesDataSetReader reads the tissues data set file and returns PixelTissues objects. Class TissuesDataSetWriter takes PixelTissues objects and writes them to the file.
- Class PlotPixel is one of several main programs that displays the results of a spin relaxometry analysis. This program generated the plot in Figure 36.9.
- Class SpinSignal provides methods for calculating the spin signal formulas (36.1), (36.3), (36.8), and (36.9).
- Class edu.rit.numeric.NonNegativeLeastSquares provides an object that finds the solution to a nonnegative linear least-squares problem.
- Class edu.rit.numeric.NonLinearLeastSquares provides an object that finds the solution to a nonlinear least-squares problem using the Levenberg-Marquardt method. This class finds a minimum of the sum of the squares of any series of functions; the functions are defined by a class implementing interface edu.rit.numeric.VectorFunction.
- Class SpinSignalDifference implements interface VectorFunction and provides methods for calculating the spin signal functions (36.7) and their partial derivatives (36.8–36.9) to be solved by class NonLinearLeastSquares.
- Class PixelAnalysis provides a static `analyze()` method that carries out the MRI spin relaxometry analysis procedure on one pixel, as described in Section 36.2.

```
package edu.rit.mri;
import edu.rit.mri.SpinSignal;
import edu.rit.mri.SpinSignalDifference;
import edu.rit.numeric.NonLinearLeastSquares;
import edu.rit.numeric.NonNegativeLeastSquares;
import edu.rit.numeric.Series;
import edu.rit.numeric.TooManyIterationsException;
import java.util.ArrayList;
import java.util.List;
public class PixelAnalysis
   {
   /**
    * Do a spin relaxometry analysis.
    *
    * @param  t_series
    *     Series of measured time values, of length M (input).
    * @param  S_series
    *     Series of measured spin signal values, of length M
    *     (input).
    * @param  R1_series
    *     Series of fixed spin-lattice relaxation rates for the
    *     linear part of the analysis, of length N (input).
    * @param  A
    *     Design matrix for the linear part of the analysis (input).
    *     This must be an MxN-element matrix such that A[i][j] =
    *     1 - 2*exp(-R1[j]*t[i]). (The design matrix is supplied as
    *     an argument because the same design matrix is typically
    *     used for every pixel in an image, and calculating the
    *     design matrix just once outside this routine saves time.)
    * @param  rho_list
    *     List in which to store the computed spin densities
    *     (output). The size of the list is the number of tissues.
    *     If the routine could not find a solution, the size of the
    *     list is 0.
    * @param  R1_list
    *     List in which to store the computed spin-lattice
    *     relaxation rates (output). The size of the list is the
    *     number of tissues. If the routine could not find a
    *     solution, the size of the list is 0.
    */
   public static void analyze
      (Series t_series,
       Series S_series,
       Series R1_series,
       double[][] A,
       List<Double> rho_list,
       List<Double> R1_list)
      {
      int M = t_series.length();
      int N = R1_series.length();

      // Do a spin relaxometry analysis using nonnegative linear
      // least squares.

      // Create nonnegative linear least squares solver.
      NonNegativeLeastSquares linsolver =
         new NonNegativeLeastSquares (M, N);

      // Find the solution.
      for (int i = 0; i < M; ++ i)
         {
         System.arraycopy (A[i], 0, linsolver.a[i], 0, N);
         linsolver.b[i] = S_series.x(i);
         }
      linsolver.solve();
      double[] rho_series = linsolver.x;

      // Find peaks in the solution. A peak occurs at index i if
      // rho[i] > rho[i-1] and rho[i] > rho[i+1].
      ArrayList<Double> approx_rho_list = new ArrayList<Double>();
      ArrayList<Double> approx_R1_list = new ArrayList<Double>();
      for (int j = 0; j < N; ++ j)
         {
         if (rho_series[j] > (j == 0 ? 0.0 : rho_series[j-1]) &&
               rho_series[j] > (j == N-1 ? 0.0 : rho_series[j+1]))
            {
            approx_rho_list.add (rho_series[j]);
            approx_R1_list.add (R1_series.x(j));
            }
         }

      // Do a spin relaxometry analysis using nonlinear least
      // squares. Peaks in the linear analysis give the initial
      // vector of densities and rates.

      // Repeat until the solution is plausible.
      boolean plausible = false;
      int L = approx_rho_list.size();
      rho_list.clear();
      R1_list.clear();
      while (L > 0 && ! plausible)
         {
```

```
// Create spin signal difference function. L = number of
// tissues.
SpinSignalDifference fcn =
   new SpinSignalDifference (t_series, S_series, L);

// Create nonlinear least squares solver.
NonLinearLeastSquares nonlinsolver =
   new NonLinearLeastSquares (fcn);

// Find the solution.
for (int i = 0; i < L; ++ i)
   {
   nonlinsolver.x[(i<<1)] = approx_rho_list.get(i);
   nonlinsolver.x[(i<<1)+1] = approx_R1_list.get(i);
   }
try
   {
   nonlinsolver.solve();
   for (int i = 0; i < L; ++ i)
      {
      rho_list.add (nonlinsolver.x[(i<<1)]);
      R1_list.add (nonlinsolver.x[(i<<1)+1]);
      }

   // Decide if solution is plausible.
   plausible =
      checkPlausibility (S_series, rho_list, R1_list);
   }

// Couldn't find a solution.
catch (TooManyIterationsException exc)
   {
   plausible = false;
   }

// If solution is not plausible, eliminate tissue with
// smallest density and try again.
if (! plausible)
   {
   double minrho = Double.MAX_VALUE;
   int mini = 0;
   for (int i = 0; i < L; ++ i)
      {
      if (approx_rho_list.get(i) < minrho)
         {
         minrho = approx_rho_list.get(i);
```

```
               mini = i;
               }
            }
         approx_rho_list.remove (mini);
         approx_R1_list.remove (mini);
         L = approx_rho_list.size();
         rho_list.clear();
         R1_list.clear();
         }
      }
   }
```

The `analyze()` method has to deal with a practical issue that doesn't rear its head until you start analyzing actual MR image data: Sometimes the nonlinear least-squares algorithm comes up with a nonsensical solution. Therefore, the `analyze()` method checks the nonlinear least-squares fit for plausibility. To be plausible, the following must be true:

- All spin densities and spin-lattice relaxation rates must be positive. If this is not the case, it's likely the nonlinear least-squares algorithm is trying to fit the data to too many parameters, resulting in nonsensical parameter values.
- All the spin-lattice relaxation rates must be sufficiently far apart. Specifically, the relative difference between any two rates must be greater than 0.001. If the rates are closer together than that, it's likely they represent the same tissue.
- The sum of the spin densities must agree with the asymptotic spin signal for large values of *t*. (In Equation 36.2, if *t* is large, then the exponential function is nearly zero, and $S(t)$ becomes the sum of the $\rho_i$ values.) Specifically, the sum of the spin densities must be within 20 percent of the average of the last seven spin signal values. If this is not the case, it's likely that two spurious "tissues," one with a very large relaxation rate and one with a very small relaxation rate, are canceling each other out, and there really should be only one tissue.

If the nonlinear least-squares fit is not plausible, the `analyze()` method decides it is trying to fit too many tissues. The `analyze()` method eliminates the tissue with the smallest spin density and repeats the nonlinear least-squares fit. This continues until the fit is plausible or until all the tissues have been eliminated, in which case the `analyze()` method reports that it could not find a solution.

```
/**
 * Decide if the given solution is plausible.
 */
private static boolean checkPlausibility
   (Series S_series,
    List<Double> rho_list,
 List<Double> R1_list)
{
int M = S_series.length();
int L = rho_list.size();

// If any density or rate is negative, solution is not
// plausible.
for (int i = 0; i < L; ++ i)
   {
   if (rho_list.get(i) < 0.0)
      {
      return false;
      }
   if (R1_list.get(i) < 0.0)
      {
      return false;
      }
   }

// If relative difference between any two rates is too small,
// solution is not plausible.
for (int i = 0; i < L-1; ++ i)
   {
   double R_i = R1_list.get(i);
   for (int j = i+1; j < L; ++ j)
      {
      double R_j = R1_list.get(j);
      double reldiff =
         2.0*Math.abs(R_i-R_j)/Math.abs(R_i+R_j);
      if (reldiff <= 0.001)
         {
         return false;
         }
      }
   }

// If sum of densities is too far from asymptotic measurement
// for large t, solution is not plausible.
double sumrho = 0.0;
for (int i = 0; i < L; ++ i)
   {
   sumrho += rho_list.get(i);
   }
double S_last = 0.0;
int n = 0;
for (int i = M-1; i >=0 && n < 7; -- i)
```

```
         {
         S_last += S_series.x(i);
         ++ n;
         }
      S_last /= n;
      double reldiff = Math.abs(sumrho-S_last)/Math.abs(S_last);
      if (reldiff >= 0.2)
         {
         return false;
         }

      // Solution is plausible.
      return true;
      }
   }
```

Finally, class SpinRelaxometrySeq is the main program class for the sequential version of the spin relaxometry analysis program. The command-line arguments are the following:

- Lower spin-lattice relaxation rate, $R_L$.
- Upper spin-lattice relaxation rate, $R_U$.
- Number of spin-lattice relaxation rate intervals, $N$. For the nonnegative linear least-squares portion of the analysis, the program uses $N$+1 relaxation rates logarithmically spaced from $R_L$ to $R_U$.
- One or more input spin signal data set file names. If an input file name is, say, "`signal01.dat`", then the corresponding output tissues data set file name is "`tissues_signal01.dat`".

Here is the source code for class SpinRelaxometrySeq.

```
package edu.rit.mri;
import edu.rit.io.Files;
import edu.rit.numeric.ArraySeries;
import edu.rit.numeric.Series;
import java.io.File;
import java.util.ArrayList;
public class SpinRelaxometrySeq
   {
   public static void main
      (String[] args)
      throws Exception
      {
```

```
// Start timing.
long t1 = System.currentTimeMillis();

// Parse command line arguments.
if (args.length < 4) usage();
double R1_lower = Double.parseDouble (args[0]);
double R1_upper = Double.parseDouble (args[1]);
int N = Integer.parseInt (args[2]);
String[] signalfilename = new String [args.length-3];
System.arraycopy (args, 3, signalfilename, 0, args.length-3);

// Compute spin relaxation rates.
double[] R1 = new double [N+1];
double log_R1_lower = Math.log (R1_lower);
double log_R1_upper = Math.log (R1_upper);
double interval = (log_R1_upper – log_R1_lower)/N;
for (int j = 0; j <= N; ++ j)
   {
   R1[j] = Math.exp (log_R1_lower + j*interval);
   }
ArraySeries R1_series = new ArraySeries (R1);

// Set up lists to receive analysis results.
ArrayList<Double> rho_list = new ArrayList<Double>();
ArrayList<Double> R1_list = new ArrayList<Double>();

// Analyze each input spin signal data set file.
for (int f = 0; f < signalfilename.length; ++ f)
   {
   File signalfile = new File (signalfilename[f]);
   File tissuesfile = new File
      (Files.fileNamePrepend
         (signalfilename[f], "tissues_"));
   // Set up data set reader and writer.
   SignalDataSetReader reader =
      new SignalDataSetReader (signalfile);
   int H = reader.getHeight();
   int W = reader.getWidth();
   TissuesDataSetWriter writer =
      new TissuesDataSetWriter (tissuesfile, H, W);
```

The program uses the same time series and design matrix, calculated here, for each pixel in the spin signal data set.

```
            // Get time series.
            Series t_series = reader.getTimeSeries();
            int M = t_series.length();

            // Compute design matrix.
            double[][] A = new double [M] [N+1];
            for (int i = 0; i < M; ++ i)
               {
               double[] A_i = A[i];
               double t_i = t_series.x(i);
               for (int j = 0; j <= N; ++ j)
                  {
                  A_i[j] = SpinSignal.S (R1[j], t_i);
                  }
               }

            // Analyze all pixels.
            int P = reader.getPixelCount();
            for (int i = 0; i < P; ++ i)
               {
               PixelSignal signal = reader.getPixelSignal (i);
               if (signal != null)
                  {
```

To do the analysis, the program uses the `analyze()` method in class PixelAnalysis, which we studied previously.

```
                  // Do the spin relaxometry analysis.
                  PixelAnalysis.analyze
                     (t_series, signal.S_measured(), R1_series, A,
                      rho_list, R1_list);
```

The program bundles the pixel's analysis results into an instance of class PixelTissues, which contains the index of the spin signal data set file, the index of the pixel within the file, the $\rho_j$ values, and the $R_j$ values. The tissues data set writer then writes the information in the PixelTissues object to the file. While this auxiliary PixelTissues object is not really necessary in the sequential version, it will be crucial in the parallel version.

```
                  // Write results to data set.
                  writer.addPixelTissues
                     (new PixelTissues (f, i, rho_list, R1_list));
                  }
               }
```

```
         // All done.
         reader.close();
         writer.close();
         }

      // Stop timing.
      long t2 = System.currentTimeMillis();
      System.out.println ((t2-t1)+" msec");
      }
   }
```

## 36.4 Cluster Parallel Program Design

To do an MRI spin relaxometry analysis on a cluster parallel computer, we'll use the same workflow as in Figure 36.10, except we'll replace the sequential analysis program with a cluster parallel analysis program.

The first design decision is how to partition the computation among the parallel processors. Because the Levenberg-Marquardt nonlinear least-squares algorithm is an iterative algorithm, different pixels, in general, require different numbers of iterations. Also, different pixels have different numbers of tissues, and the more tissues there are, the longer it takes to calculate the model functions (36.7) and their partial derivatives (36.8–36.9). Therefore, different pixels' computations will different amounts of time, and load balancing is required. As we have done with other cluster parallel programs, we will use the *master-worker pattern* to balance the load.

The master divides the computation into chunks and sends the chunks to the workers. However, this time, the chunk is not simply a Range object. For this program, a chunk of computation consists of the file index (designating one of the spin signal data set files on the command line), the index of the first pixel to analyze within that file, and the number of consecutive pixels to analyze. Class PixelChunk encapsulates this information in an object; the master sends pixel chunk objects to the workers. To do the partitioning, the master uses an instance of class PixelSchedule. The master initializes the pixel schedule object with the names of the input spin signal data set files from the command line. The pixel schedule object reads the files (using class SignalDataSetReader) to determine the number of pixels in each data set. The master repeatedly calls the pixel schedule's `next()` method, which returns a sequence of pixel chunks, each chunk consisting of 100 pixels from one of the spin signal data sets.

The second design decision is how the workers will obtain the spin signal data for the pixels the master tells the workers to analyze. We will use the *parallel input files pattern.* Each worker can read all of the spin signal data set files. When told by the master to analyze a series of pixels in a certain file, the worker uses class SignalDataSetReader to read the file directly.

The third design decision is how the program will write the output tissues data set files. We will let the master do all the output file writing. After analyzing a chunk of pixels, the worker sends a group of PixelTissues objects to the master. As we have seen already, the PixelTissues object contains the index of the spin signal data set file, the index of the pixel within the file, the $\rho_j$ values, and the $R_j$ values. The master uses the file index to select the proper output file, and then writes the tissues data to that file. Although this approach does require communicating the analysis results from the workers to the master, the amount of communication is minor compared to the amount of computation.

Even though the master is doing somewhat more work (writing the output files) than in other master-worker programs we have studied, typically the master is still not doing enough work to keep its CPU fully occupied. Therefore, we still want both a master thread and a worker thread in process 0. The worker thread can utilize the CPU to analyze pixels while the master thread is waiting to receive the next group of analysis results.

Figure 36.12 shows the overall execution timeline of the cluster parallel MRI spin relaxometry analysis program.

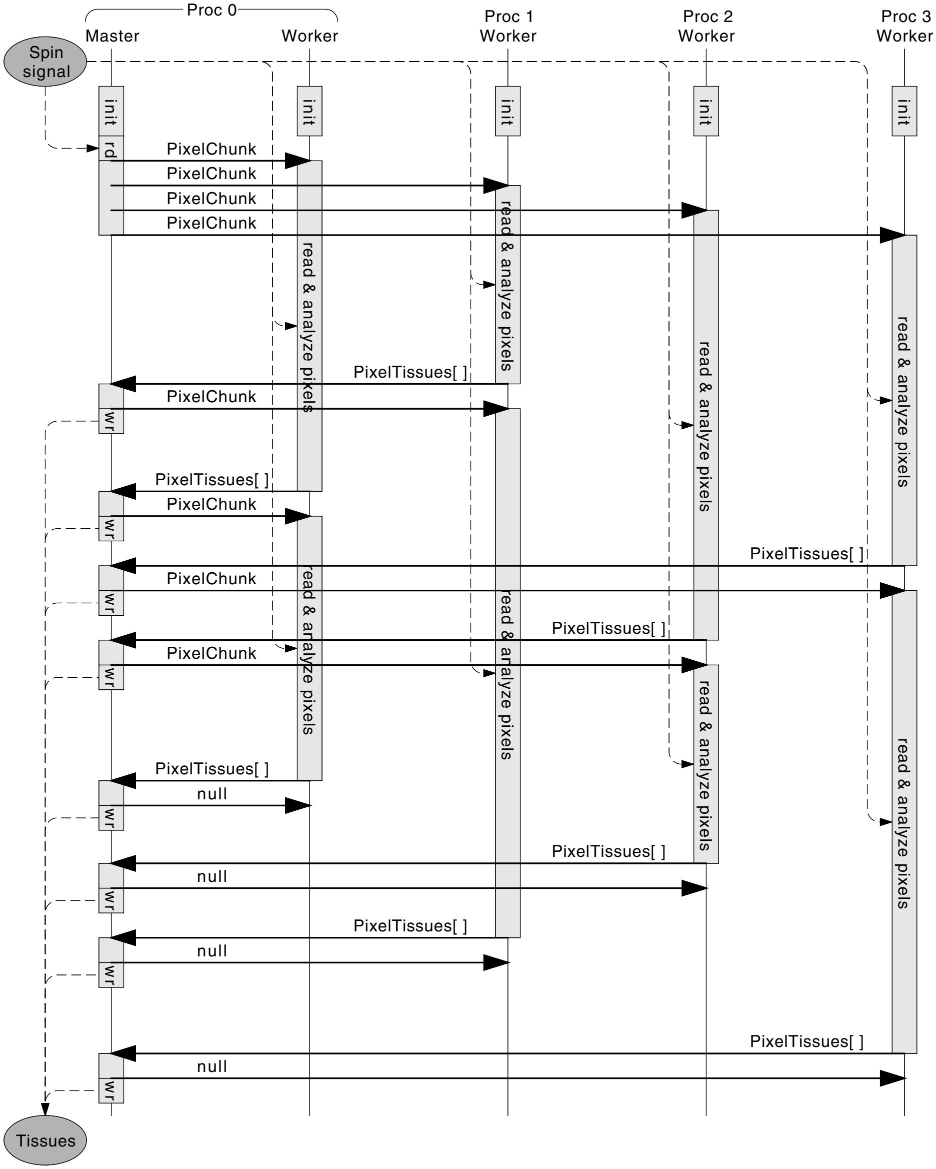


**Figure 36.12** Cluster parallel MRI spin relaxometry analysis program execution timeline

## 36.5 Parallel Spin Relaxometry Program

Before looking at the parallel version of the MRI spin relaxometry analysis program, let's look at class PixelChunk, which the master uses to send chunks of work to the workers. As discussed in Chapter 22, because this is an object to be sent in a message, we must make its class serializable. There are two ways to make a class serializable. One way is simply to declare that the class implements interface java.io.Serializable. The other way is to declare that the class implements interface java.io.Externalizable. For a Serializable class, we don't have to do anything further; the Java platform automatically converts the object to and from a stream of bytes when necessary. For an Externalizable class, the Java platform does not do this automatically, and we must write the serialization and deserialization code ourselves. On the other hand, the number of bytes in the serialized form of a Serializable object is typically more than—sometimes several times as many as—the number of bytes in the serialized form of an Externalizable object. To minimize the time required to send messages containing objects, we usually want to make the objects' classes Externalizable. Here is the code for class PixelChunk.

```
package edu.rit.mri;
import java.io.Externalizable;
import java.io.IOException;
import java.io.ObjectInput;
import java.io.ObjectOutput;
public class PixelChunk
   implements Externalizable
   {
   private int fileIndex;
   private int pixelIndex;
   private int pixelCount;

   /**
    * Construct a new, uninitialized pixel chunk object. This
    * constructor is for use only by object deserialization.
    */
   public PixelChunk()
      {
      }

   /**
    * Construct a new pixel chunk object.
    */
   public PixelChunk
      (int fileIndex,
       int pixelIndex,
       int pixelCount)
      {
      this.fileIndex = fileIndex;
      this.pixelIndex = pixelIndex;
      this.pixelCount = pixelCount;
      }

   /**
    * Get the file index.
    */
   public int fileIndex()
      {
      return fileIndex;
      }

   /**
    * Get the pixel index of the first pixel to analyze.
    */
   public int pixelIndex()
      {
      return pixelIndex;
      }

   /**
    * Get the number of pixels to analyze.
    */
   public int pixelCount()
      {
      return pixelCount;
      }
```

The `writeExternal()` method converts an instance of class PixelChunk to its serialized representation by writing each field to the object output stream as a four-byte integer.

```
      /**
       * Write this pixel chunk object to the given object output
       * stream.
       */
      public void writeExternal
         (ObjectOutput out)
         throws IOException
         {
         out.writeInt (fileIndex);
         out.writeInt (pixelIndex);
         out.writeInt (pixelCount);
         }
```

The `readExternal()` method sets the state of an instance of class PixelChunk from its serialized representation by reading each field from the object input stream as a four-byte integer.

```
/**
 * Read this pixel chunk object from the given object input
 * stream.
 */
public void readExternal
   (ObjectInput in)
   throws IOException
   {
   fileIndex = in.readInt();
   pixelIndex = in.readInt();
   pixelCount = in.readInt();
   }
}
```

Similarly, class PixelTissues, which the workers use to send the analysis results back to the master, implements interface Externalizable and provides the `writeExternal()` and `readExternal()` methods.

Here is the source code for class SpinRelaxometryClu, the cluster parallel MRI spin relaxometry analysis program. Its command-line arguments are the same as the sequential version.

```
package edu.rit.mri;
import edu.rit.mp.ObjectBuf;
import edu.rit.mp.buf.ObjectItemBuf;
import edu.rit.numeric.ArraySeries;
import edu.rit.numeric.Series;
import edu.rit.pj.Comm;
import edu.rit.pj.CommStatus;
import edu.rit.pj.IntegerSchedule;
import edu.rit.pj.ParallelRegion;
import edu.rit.pj.ParallelSection;
import edu.rit.pj.ParallelTeam;
import edu.rit.util.Range;
import java.io.File;
import java.io.IOException;
import java.util.ArrayList;
public class SpinRelaxometryClu
   {
   // World communicator.
   static Comm world;
   static int size;
   static int rank;
```

```
// Command line arguments.
static double R1_lower;
static double R1_upper;
static int N;
static String[] signalfilename;

/**
 * Main program.
 */
public static void main
   (String[] args)
   throws Exception
   {
   // Start timing.
   long t1 = System.currentTimeMillis();

   // Initialize world communicator.
   Comm.init (args);
   world = Comm.world();
   size = world.size();
   rank = world.rank();

   // Parse command line arguments.
   if (args.length < 4) usage();
   R1_lower = Double.parseDouble (args[0]);
   R1_upper = Double.parseDouble (args[1]);
   N = Integer.parseInt (args[2]);
   signalfilename = new String [args.length-3];
   System.arraycopy (args, 3, signalfilename, 0, args.length-3);
```

The main program follows the standard pattern of a two-thread parallel team executing the master and worker sections in process 0, and a single thread executing just the worker section in processes 1 and up.

```
   // Process 0 executes the master section and the worker
   // section in parallel threads.
   if (rank == 0)
      {
      new ParallelTeam(2).execute (new ParallelRegion()
         {
         public void run() throws Exception
            {
            execute (new ParallelSection()
               {
               public void run() throws Exception
```

```
                     {
                     masterSection();
                     }
                  },
               new ParallelSection()
                  {
                  public void run() throws Exception
                     {
                     workerSection();
                     }
                  });
               }
            });
         }

      // Processes 1 and up execute just the worker section.
      else
         {
         workerSection();
         }

      // Stop timing.
      long t2 = System.currentTimeMillis();
      System.out.println ((t2-t1)+" msec "+rank);
      }
```

Here is the master section.

```
   /**
    * Execute the master section.
    */
   private static void masterSection()
      throws IOException
      {
      int worker;
      PixelChunk chunk;
```

The master section begins by creating a tissues data set writer for every output file. Because the workers could return analysis results for the various data sets in any order, the master must be prepared to write analysis results to any of the output files.

```
// Set up array of writers for output tissues data sets.
TissuesDataSetWriter[] writer =
   new TissuesDataSetWriter [signalfilename.length];
for (int f = 0; f < writer.length; ++ f)
   {
   SignalDataSetReader reader =
      new SignalDataSetReader (new File (signalfilename[f]));
   int H = reader.getHeight();
   int W = reader.getWidth();
   reader.close();
   writer[f] =
      new TissuesDataSetWriter
         (new File (“tissues.”+signalfilename[f]), H, W);
   }
```

The master uses an instance of class PixelSchedule to partition the computation into chunks.

```
// Set up schedule to analyze pixels in chunks of 100.
PixelSchedule schedule =
   new PixelSchedule (100, signalfilename);

// Send initial chunk to each worker. If null, no more work
// for that worker. Keep count of active workers.
int activeWorkers = size;
for (worker = 0; worker < size; ++ worker)
   {
   chunk = schedule.next();
   world.send (worker, ObjectBuf.buffer (chunk));
   if (chunk == null) -- activeWorkers;
   }

// Repeat until all workers have finished.
while (activeWorkers > 0)
   {
```

To receive analysis results from a worker, any worker, the master sets up a buffer for a single object of type “array of pixel tissues” (`PixelTissues[]`). This lets the master receive an array of any length without needing to know ahead of time how many array elements the worker will send.

```
   // Receive a chunk of pixel tissues from any worker.
   ObjectItemBuf<PixelTissues[]> buf = ObjectBuf.buffer();
   CommStatus status = world.receive (null, buf);
   worker = status.fromRank;
```

```
         // Send next chunk to that specific worker. If null, no
         // more work.
         chunk = schedule.next();
         world.send (worker, ObjectBuf.buffer (chunk));
         if (chunk == null) -- activeWorkers;
```

The master takes however many pixel tissues objects the worker sent and writes each one of them to the appropriate output file. The file index comes from the pixel tissues object itself.

```
         // Record pixel tissues in output tissues data set.
         for (PixelTissues tissues : buf.item)
            {
            if (tissues != null)
               {
               writer[tissues.fileIndex()]
                  .addPixelTissues (tissues);
               }
            }
         }

      // All done.
      for (int f = 0; f < writer.length; ++ f)
         {
         writer[f].close();
         }
      }
```

Here is the worker section.

```
   /**
    * Execute the worker section.
    */
   private static void workerSection()
      throws IOException
      {
      int fileIndex = -1;
      SignalDataSetReader reader = null;
      Series t_series = null;
      int M = 0;
      double[][] A = null;

      // Compute spin relaxation rates.
      double[] R1 = new double [N+1];
      double log_R1_lower = Math.log (R1_lower);
      double log_R1_upper = Math.log (R1_upper);
      double interval = (log_R1_upper - log_R1_lower)/N;
      for (int j = 0; j <= N; ++ j)
         {
         R1[j] = Math.exp (log_R1_lower + j*interval);
         }
      ArraySeries R1_series = new ArraySeries (R1);

      // Set up lists to receive analysis results.
      ArrayList<Double> rho_list = new ArrayList<Double>();
      ArrayList<Double> R1_list = new ArrayList<Double>();

      // Repeat until no more work.
      workerloop: for (;;)
         {
         // Receive a chunk of pixel indexes from the master. If
         // null, no more work.
         ObjectItemBuf<PixelChunk> buf = ObjectBuf.buffer();
         world.receive (0, buf);
         PixelChunk chunk = buf.item;
         if (chunk == null) break workerloop;
         int f = chunk.fileIndex();
         int lb = chunk.pixelIndex();
         int len = chunk.pixelCount();
```

The worker must be prepared to analyze pixels in any of the input files, as directed by the master. The worker uses the `fileIndex` variable to remember which input file the worker is currently reading. If the next pixel chunk refers to a different file index, the worker switches to the new input file.

```
         // If we are now working on a different file:
         if (f != fileIndex)
            {
            // Close old file.
            if (reader != null) reader.close();

            // Open new file.
            fileIndex = f;
            reader =
               new SignalDataSetReader
                  (new File (signalfilename[f]));

            // Get time series.
            t_series = reader.getTimeSeries();
            M = t_series.length();
            // Compute design matrix.
            A = new double [M] [N+1];
            for (int i = 0; i < M; ++ i)
               {
               double[] A_i = A[i];
               double t_i = t_series.x(i);
               for (int j = 0; j <= N; ++ j)
                  {
                  A_i[j] = SpinSignal.S (R1[j], t_i);
                  }
               }
            }

         // Set up array of pixel tissues to hold analysis results.
         PixelTissues[] tissues = new PixelTissues [len];

         // Process all pixels in chunk.
         for (int i = 0; i < len; ++ i)
            {
            int index = lb + i;
            PixelSignal signal_i = reader.getPixelSignal (index);
            if (signal_i != null)
               {
               // Get measured spin signal.
               Series S_series = signal_i.S_measured();

               // Do the spin relaxometry analysis.
               PixelAnalysis.analyze
                  (t_series, S_series, R1_series, A,
                   rho_list, R1_list);

               // Record analysis results.
               tissues[i] =
                  new PixelTissues (f, index, rho_list, R1_list);
               }
            }
```

At this point, the `tissues` variable is an array of the analysis results for the pixels in the chunk. The worker must send this array as a single object of type "array of pixel tissues," because that is what the master expects to receive. To set up the proper source buffer, the worker calls the `objectBuffer()` method of class ObjectBuf; this method returns a buffer that sends the entire array as a single object. (In contrast, the `buffer()` method of class ObjectBuf returns a buffer that sends the array elements individually, as separate objects. If the worker did that, the master would need to know the number of array elements ahead of time.)

```
            // Send chunk of pixel tissues to the master.
            world.send (0, ObjectBuf.objectBuffer (tissues));
            }

         if (reader != null) reader.close();
         }
      }
```

## 36.6 Parallel Program Performance

Table 36.1 lists, and Figure 36.13 plots, the SpinRelaxometryClu program's performance on the "tardis" parallel computer. The input was a 64 × 64 × 64-voxel (a 256K-voxel) MR image with 64 spin signal samples for each voxel. Each slice was a copy of the 64 × 64-pixel example image we've been using. For the nonnegative linear least-squares curve fit, the program used 200 relaxation rates logarithmically spaced from 0.01 to 10.0. The program experienced efficiencies of 90 percent or better, out to 14 processes.

**Table 36.1** SpinRelaxometrySeq/Clu running-time metrics

| *N* | *K* | *T* | *Spdup* | *Eff* | *EDSF* |
|---:|---:|---:|---:|---:|---:|
| 256K | seq | 829842 | | | |
| 256K | 1 | 781801 | 1.061 | 1.061 | |
| 256K | 2 | 403596 | 2.056 | 1.028 | 0.032 |
| 256K | 3 | 272539 | 3.045 | 1.015 | 0.023 |
| 256K | 4 | 211635 | 3.921 | 0.980 | 0.028 |
| 256K | 5 | 170687 | 4.862 | 0.972 | 0.023 |
| 256K | 6 | 144597 | 5.739 | 0.956 | 0.022 |
| 256K | 8 | 109578 | 7.573 | 0.947 | 0.017 |
| 256K | 10 | 89517 | 9.270 | 0.927 | 0.016 |
| 256K | 14 | 65088 | 12.750 | 0.911 | 0.013 |
| 256K | 20 | 47133 | 17.606 | 0.880 | 0.011 |
| 256K | 28 | 38299 | 21.667 | 0.774 | 0.014 |
| 256K | 40 | 31301 | 26.512 | 0.663 | 0.015 |

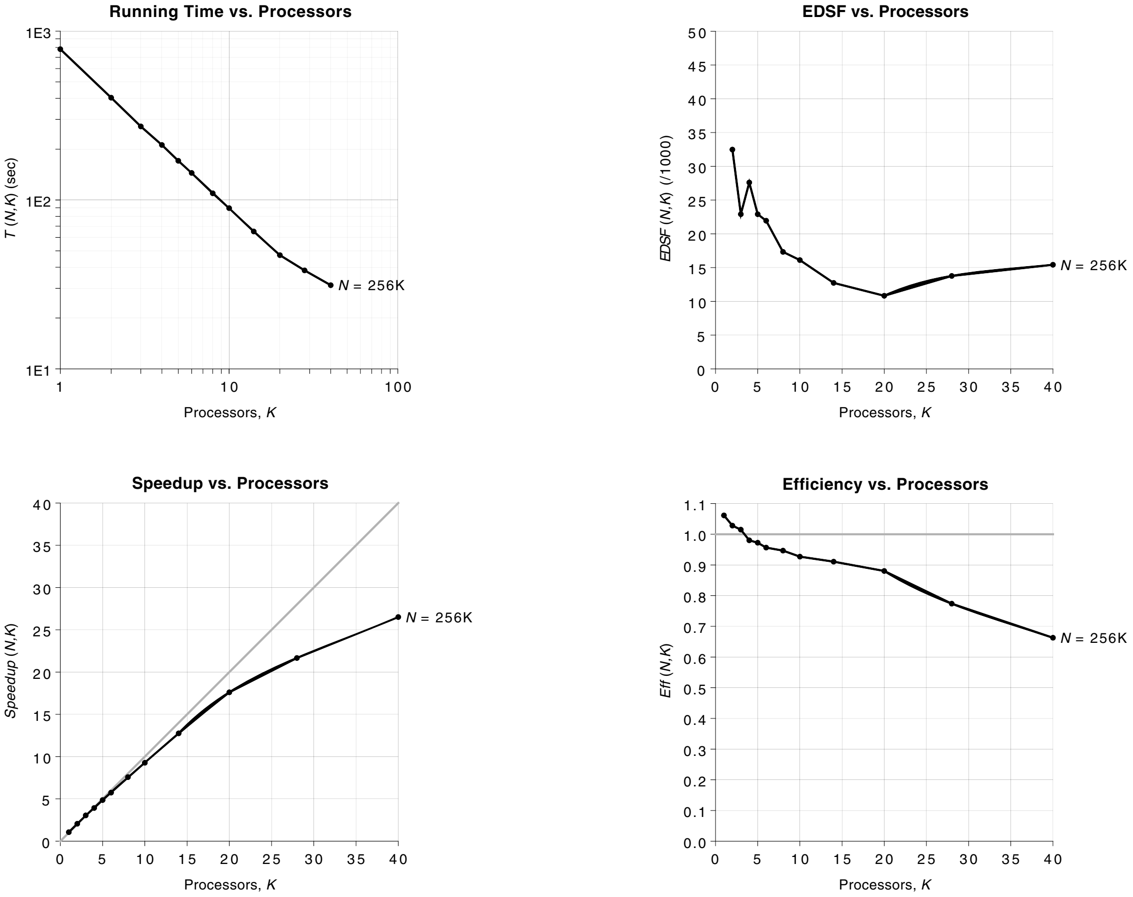


**Figure 36.13** SpinRelaxometrySeq/Clu running-time metrics

One impediment to widespread usage of MRI spin relaxometry has been the amount of computer time needed to do the analysis. However, because the computations for each pixel are independent, spin relaxometry analysis is an ideal candidate for speedup on a parallel computer. With 64 × 64-voxel slices, the SpinRelaxometryClu program took half a minute to analyze the MR image on 40 processors—about 5 milliseconds per voxel. If we scaled each slice's size up by a factor of 16 and analyzed a 256 × 256 × 64-voxel MR image, we'd expect the program to take 8 minutes on 40 processors, or even less time on more processors. That ought to be fast enough to make MRI spin relaxometry analysis a useful diagnostic tool.

## 36.7 Displaying the Results

The last stage of the MRI spin relaxometry analysis workflow (Figure 36.10) is to display the results. We've already seen one example: a plot of the measured spin signal and the fitted curve for a certain pixel (Figure 36.9). Program edu.rit.mri.PlotPixel generated this plot.

More interesting would be a summary of the results for the whole MR image, not just one pixel. Because the spin-lattice relaxation rates reflect the chemical composition of the tissues imaged, a useful summary display is a histogram of the relaxation rates (Figure 36.14). Program edu.rit.mri.R1Histogram generated these plots from the computed tissues data set for one slice of the example MR image. The first plot has 100 histogram bins of width 0.01 from $R = 0.0$ to 1.0. That is, the first bin counts the number of pixels having a tissue with relaxation rate $0.00 \le R < 0.01$; the second bin counts the number of pixels having a tissue with relaxation rate $0.01 \le R < 0.02$; and so on. The second plot has 100 histogram bins of width 0.01 from $R = 1.0$ to 2.0. A peak in the histogram at a relaxation rate not normally encountered in healthy tissue may indicate the presence of abnormal chemicals and lead to a diagnosis of disease.

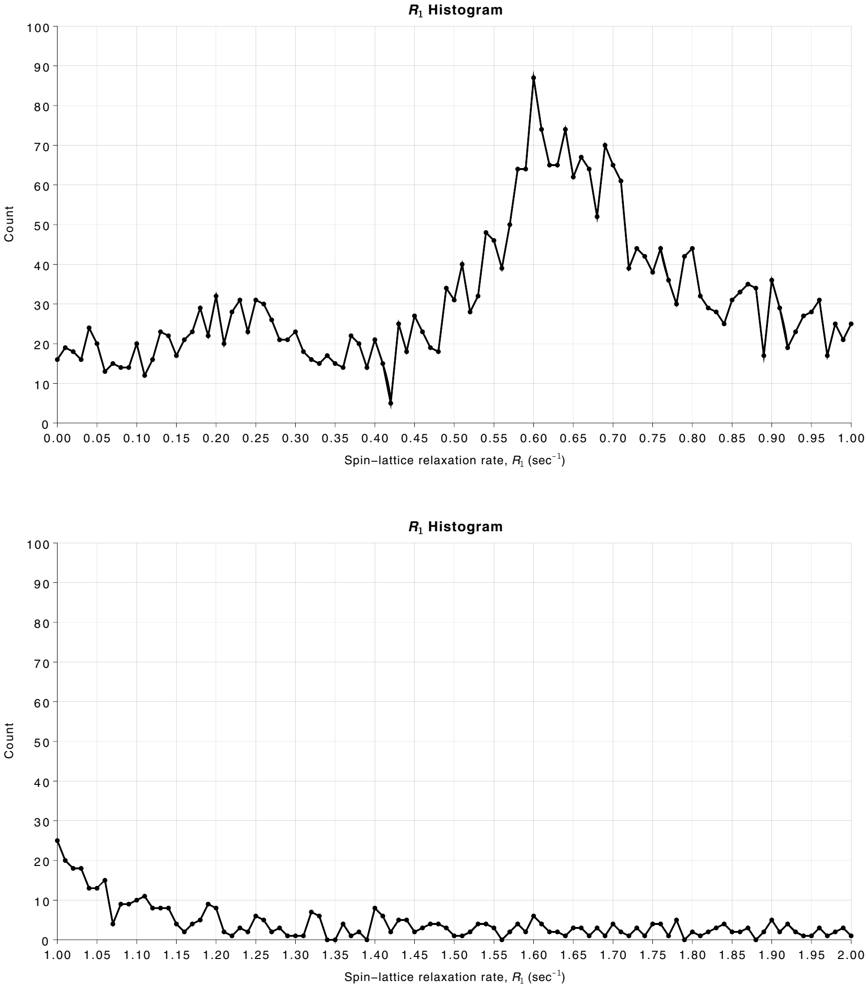
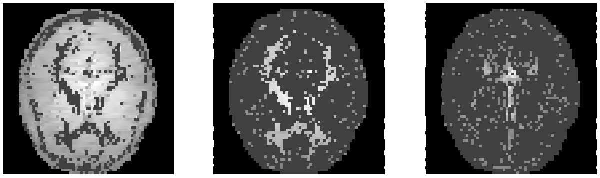


**Figure 36.14** Histograms of the spin-lattice relaxation rates

Also of interest are the locations of tissues having certain spin-lattice relaxation rates. If the doctor notices a spike in the histogram at an abnormal relaxation rate, the location of the possibly diseased tissues that engendered the spike may help the doctor diagnose the illness.

Figure 36.15 shows some examples of images depicting tissues with certain spin-lattice relaxation rates. Program edu.rit.mri.R1Image generated these images from the computed tissues data set for one slice of the example MR image. A pixel with no tissues with $R$ in the given range is colored dark gray. A pixel with one or more tissues with $R$ in the given range is colored medium gray to white, with medium gray corresponding to a small spin density and white corresponding to a large spin density. Thus, the brighter the pixel, the higher the concentration of atoms with $R$ in the given range. The first image depicts the range of $R$ values expected for gray matter (glial cells and neuron cell bodies in the brain). The gray matter is distributed throughout the brain, except for certain regions in the middle, and except for the bones of the cranium (the thin gray region around the periphery). The second image depicts the range of $R$ values expected for white matter (primarily axons, the fibers that interconnect neurons). The white matter has a high concentration in the regions where the gray matter has a low concentration. There's a hump in the histogram at about $R$=0.15–0.30; this is probably an accumulation of cerebrospinal fluid. The third image shows that these tissues are located mainly in the center of the brain, in the fissure between the two cerebral hemispheres.



$0.47 \le R < 0.92$ $0.93 \le R < 0.32$ $0.15 \le R < 0.30$

**Figure 36.15** Locations of tissues with certain spin-lattice relaxation rates

These postprocessing programs all follow the *data-set querying pattern.* Each program examines the information in a data set—namely, the spin signal input files and the tissues output files for the MR image—to answer a query. Because these particular queries do not require a lot of computation, the postprocessing programs all run quickly, and there is no point in designing them as parallel programs. Other, more complicated queries may be able to take advantage of *parallel* data-set querying programs.

## 36.8 Acknowledgments

I would like to thank Joseph Hornak in the Chester F. Carlson Center for Imaging Science at the Rochester Institute of Technology for drawing my attention to the MRI spin relaxometry analysis problem. The spin signal data set used as an example in this chapter is courtesy of C. Springer and X. Li in the Department of Chemistry at the State University of New York, Stony Brook.

## 36.9 For Further Information

On magnetic resonance imaging:

- J. Hornak. *The Basics of MRI*. 2007.
  http://www.cis.rit.edu/htbooks/mri/index.html

On MR imaging of brain tissues, including identifying brain tissues by their characteristic spin-lattice relaxation rates, spin-spin relaxation rates, and spin densities:

- L. Fletcher, J. Barsotti, and J. Hornak. A multispectral analysis of brain tissues. *Magnetic Resonance in Medicine*, 29:623–630, 1993.

On the nonnegative linear least-squares algorithm:

- C. Lawson and R. Hanson. *Solving Least Squares Problems*. Society for Industrial and Applied Mathematics, 1995.
- Netlib Repository. http://www.netlib.org/
- NNLS program. http://www.netlib.org/lawson-hanson/all

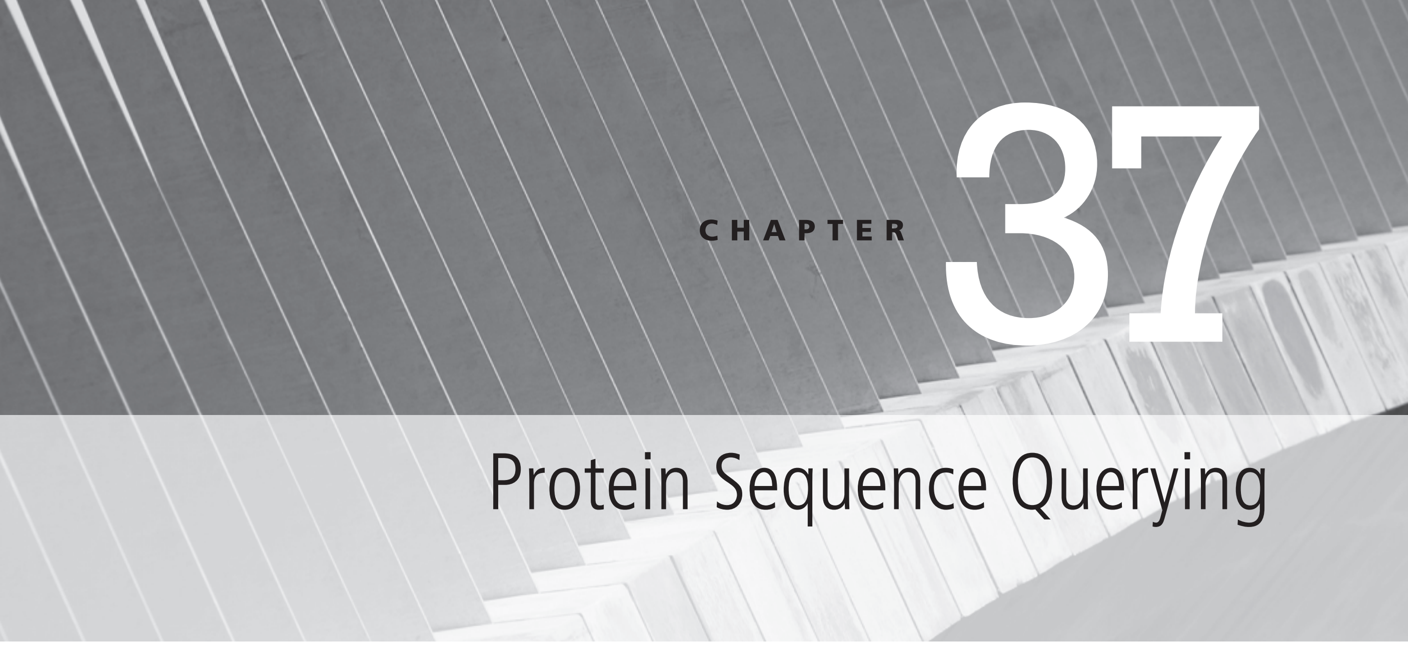
On the Levenberg-Marquardt nonlinear least-squares algorithm:

- W. Press, S. Teukolsky, W. Vetterling, and B. Flannery. *Numerical Recipes: The Art of Scientific Computing*, Third Edition. Cambridge University Press, 2008, Chapter 15.5.
- Netlib Repository. http://www.netlib.org/
- MINPACK. http://www.netlib.org/minpack/

On parallel programs for MRI spin relaxometry analysis:

- A. Bak, J. Hornak, and N. Schaller. From impractical to practical: solving an MRI problem using parallelism. In *Rochester Institute of Technology B. Thomas Golisano College of Computing and Information Sciences 2005 Conference on Computing and Information Sciences*, 2005.
  http://hdl.handle.net/1850/423

*This page intentionally left blank*



CHAPTER 37

# Protein Sequence Querying

in which we encounter protein sequences and databases thereof; we learn an algorithm for local alignment of protein sequences; we design two versions of a parallel protein sequence data-set querying program; and we observe the performance implications of each design

# CHAPTER 37 Protein Sequence Querying

## 37.1 Protein Sequences

**Proteins** are the fundamental building blocks of living organisms. They perform numerous and diverse functions. For example, **actin** is a component of the cytoskeleton that maintains each cell's shape. **Insulin** is a hormone that triggers cells to absorb glucose from the bloodstream. **Pepsin** is an enzyme that takes part in digesting food. **Rhodopsin**, found in the rod cells of the eye, enables vision in low light conditions.

A protein molecule consists of a number of **amino acids** chemically bonded into a long string. Here are the 20 amino acids and the letters that symbolize each one:

| | | | |
|---|---|---|---|
| A | Alanine | M | Methionine |
| C | Cysteine | N | Asparagine |
| D | Aspartic acid | P | Proline |
| E | Glutamic acid | Q | Glutamine |
| F | Phenylalanine | R | Arginine |
| G | Glycine | S | Serine |
| H | Histidine | T | Threonine |
| I | Isoleucine | V | Valine |
| K | Lysine | W | Tryptophan |
| L | Leucine | Y | Tyrosine |

A **protein sequence** is just a string of the preceding 20 letters that gives the sequence of amino acids in the protein. For example, here is the sequence of human insulin: FVNQHLCGSHLVEALYLVCGERGFFYTPKTGIVEQCCTSICSLYQLENYCN. Insulin has 51 amino acids—there are 51 letters in the sequence.

A protein "folds" into a specific three-dimensional shape that depends on the exact sequence of the amino acids. The protein's shape in turn determines the protein's function. Thus, biologists are keenly interested in analyzing and comparing protein sequences as a means to understand proteins' functions. The advent of automated techniques for determining proteins' sequences, computer databases to store protein sequences, and computer algorithms to find proteins with similar sequences has revolutionized the field and has given biologists new tools for their study of life.

In this chapter, we will attack the problem of finding all proteins in a database that are similar to a given protein. This is a *data-set querying problem,* as we studied in Chapter 35. However, the query can be performed in parallel. We are going to develop two *parallel* data-set querying programs embodying two different approaches to the problem. Before we can do that, we need to look at how to compare protein sequences for similarity.

## 37.2 Protein Sequence Alignment

When comparing two protein sequences, one sequence is called the **subject sequence**; this typically is one sequence from a protein sequence database. The other sequence is called the **query sequence**; this is typically the sequence we are trying to find in the database. Figure 37.1 shows an example of a (very short) query sequence and subject sequence.

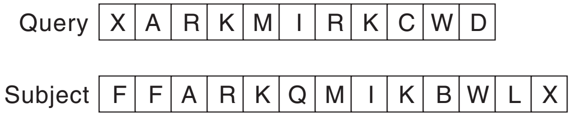

Query X A R K M I R K C W D
Subject F F A R K Q M I K B W L X

**Figure 37.1** Query sequence and subject sequence

To determine the similarity between two proteins, we will **align** the protein sequences. A **global alignment** aligns the *entire* query sequence with the *entire* subject sequence. A **local alignment** aligns a *piece* of the query sequence with a *piece* of the subject sequence. For our protein sequence database-querying problem, we are interested in local alignments. If a large enough piece of the query sequence aligns with a large enough piece of the subject sequence, that is still an interesting result, even if the entire sequences don't align. Figure 37.2 shows an example of a local alignment. The query sequence has been shifted relative to the subject sequence to align three letters. (The dimmed letters are not part of the alignment.)

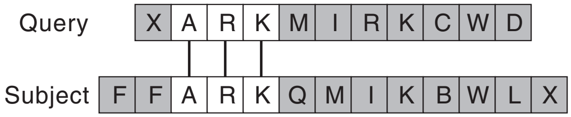

Query X A R K M I R K C W D
Subject F F A R K Q M I K B W L X

**Figure 37.2** A local alignment

An alignment has a **score** to indicate how good or bad the alignment is. For now, we will use the following simple scoring function. (Later, we will switch to a more complicated scoring function.) For each matching position, a quantity $\alpha > 0$ is added to the score. For each mismatched position, a quantity $\beta < 0$ is added to the score. For the alignment in Figure 37.2, with $\alpha = +2$ and $\beta = -1$, the score is +6.

It may be possible to increase the alignment score by introducing **gaps**. For each gap position, a quantity $\gamma < 0$ is added to the score as a gap penalty. Figure 37.3 shows another local alignment of the two example sequences including one gap; with $\gamma = -1$, the score is +9.

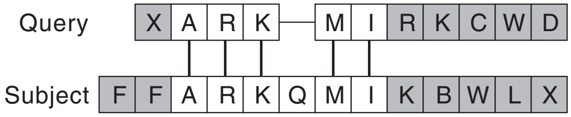

Query X A R K M I R K C W D
Subject F F A R K Q M I K B W L X

**Figure 37.3** A local alignment with a gap

Some combination of shifts and gaps will result in the largest possible alignment score. Figure 37.4 shows the best possible—the highest-scoring—local alignment of the two example sequences. With seven matching positions, one mismatched position, and two gap positions, the score is +11.

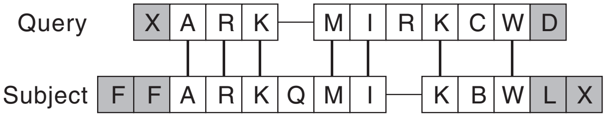


**Figure 37.4** Best possible local alignment

Now we need a way to find the best possible local alignment, given the query sequence, the subject sequence, and the scoring parameters $(\alpha, \beta, \gamma)$. In 1981, Temple Smith and Michael Waterman published an algorithm to do just that. The **Smith-Waterman algorithm** is based on the following observation. Let $A$ be the query sequence, $a_i$ be the $i$-th letter in the query sequence, $B$ be the subject sequence, $b_j$ be the $j$-th letter in the subject sequence, and $S[i, j]$ be the best possible alignment score when $a_i$ is aligned with $b_j$. Suppose that we have already found the best local alignment up through, but not including, letters $a_i$ and $b_j$. Then we have four choices for extending the alignment:

- We can align query letter $a_i$ with subject letter $b_j$. If we do, the score becomes $S[i, j] = S[i–1, j–1] + \delta(a_i, b_j)$, where $\delta(a_i, b_j) = \alpha$ if $a_i = b_j$ (a match) and $\delta(a_i, b_j) = \beta$ if $a_i \neq b_j$ (a mismatch).
- We can align query letter $a_i$ with a gap in $B$. If we do, the score becomes $S[i, j] = S[i–1, j] + \gamma$.
- We can align subject letter $b_j$ with a gap in $A$. If we do, the score becomes $S[i, j] = S[i, j–1] + \gamma$.
- We can decide that $a_i$ and $b_j$ are not part of the local alignment. If we do, the score becomes $S[i, j] = 0$.

We make the choice that results in the highest value for $S[i, j]$.

To find the best possible local alignment, we must compute $S[i, j]$ for all $i$ and $j$. The easiest way to do this is to set up a matrix of $S$ values, with $i$ indexing the rows and $j$ indexing the columns, and fill in the matrix entries. When we need a prior $S$ value, we simply look it up in the matrix. If we fill in the matrix by rows from top to bottom, and within each row by columns from left to right, all the matrix entries we need will already be filled in when we get to entry $S[i, j]$. The pseudocode follows:

```
$A[1..M] \leftarrow$ Query sequence
$B[1..N] \leftarrow$ Subject sequence
Create scoring matrix $S[0..M, 0..N]$
(All elements in row 0 of $S$) $\leftarrow 0$
(All elements in column 0 of $S$) $\leftarrow 0$
For $i = 1$ to $M$:
    For $j = 1$ to $N$:
        If $a_i = b_j$:
            $\delta \leftarrow \alpha$
        Else:
            $\delta \leftarrow \beta$
            $S[i, j] \leftarrow \max(S[i-1, j-1] + \delta, S[i-1, j] + \gamma, S[i, j-1] + \gamma, 0)$
```

Figure 37.5 shows the filled-in scoring matrix for the example sequences with scoring parameters ($\alpha$, $\beta$, $\gamma$) = (+2, –1, –1).

Subject Sequence

Query Sequence

| | | F | F | A | R | K | Q | M | I | K | B | W | L | X | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| X | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2 | 1 |
| A | 0 | 0 | 0 | 2 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 2 |
| R | 0 | 0 | 0 | 1 | 4 | 3 | 2 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 3 |
| K | 0 | 0 | 0 | 0 | 3 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 0 | 0 | 4 |
| M | 0 | 0 | 0 | 0 | 2 | 5 | 5 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 5 |
| I | 0 | 0 | 0 | 0 | 1 | 4 | 4 | 6 | 9 | 8 | 7 | 6 | 5 | 4 | 6 |
| R | 0 | 0 | 0 | 0 | 2 | 3 | 3 | 5 | 8 | 8 | 7 | 6 | 5 | 4 | 7 |
| K | 0 | 0 | 0 | 0 | 1 | 4 | 3 | 4 | 7 | 10 | 9 | 8 | 7 | 6 | 8 |
| C | 0 | 0 | 0 | 0 | 0 | 3 | 3 | 3 | 6 | 9 | 9 | 8 | 7 | 6 | 9 |
| W | 0 | 0 | 0 | 0 | 0 | 2 | 2 | 2 | 5 | 8 | 8 | 11 | 10 | 9 | 10 |
| D | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 4 | 7 | 7 | 10 | 10 | 9 | 11 |
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | |

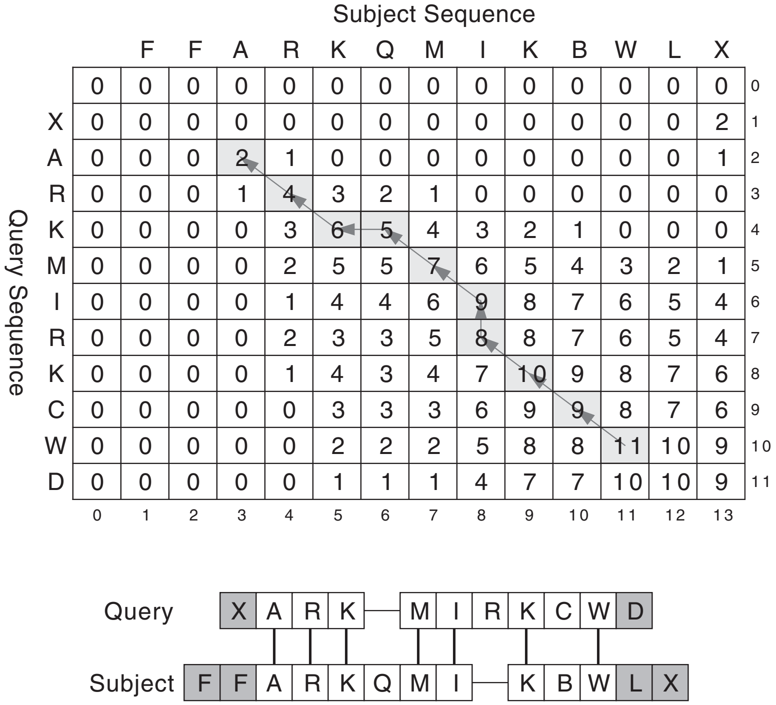
**Figure 37.5** Scoring matrix

The largest entry in the scoring matrix gives the score of the best local alignment. In Figure 37.5, the best score is +11, and it happens when query letter $a_{10}$ is aligned with subject letter $b_{11}$. Now we need to figure out the rest of the alignment—which query letters are aligned with which subject letters, and where the gaps are, if any. To do that, we will do a **traceback** and work our way from the end of the alignment to the beginning. At each step, we check which alignment choice resulted in the scoring matrix entry at the current position, and then we move backward based on that choice, stopping when we reach a zero entry. The pseudocode follows:

```
$(i, j) \leftarrow$ (row, column) of largest entry in $S$
While $S[i, j] \neq 0$:
    If $a_i = b_j$:
        $\delta \leftarrow \alpha$
    Else:
        $\delta \leftarrow \beta$
    If $S[i, j] = S[i-1, j-1] + \delta$:
        Query letter $a_i$ is aligned with subject letter $b_j$
        $(i, j) \leftarrow (i-1, j-1)$
    Else if $S[i, j] = S[i-1, j] + \gamma$:
        Query letter $a_i$ is aligned with a gap in $B$
        $(i, j) \leftarrow (i-1, j)$
```

```
Else:
    Subject letter $b_j$ is aligned with a gap in $A$
    $(i, j) \leftarrow (i, j–1)$
```

Figure 37.6 depicts the traceback for the example sequences, along with the resulting local alignment.


Subject Sequence
Query Sequence
Query X A R K — M I R K C W D
Subject F F A R K Q M I — K B W L X

| | | F | F | A | R | K | Q | M | I | K | B | W | L | X | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| X | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2 | 1 |
| A | 0 | 0 | 0 | 2 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 2 |
| R | 0 | 0 | 0 | 1 | 4 | 3 | 2 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 3 |
| K | 0 | 0 | 0 | 0 | 3 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 0 | 0 | 4 |
| M | 0 | 0 | 0 | 0 | 2 | 5 | 5 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 5 |
| I | 0 | 0 | 0 | 0 | 1 | 4 | 4 | 6 | 9 | 8 | 7 | 6 | 5 | 4 | 6 |
| R | 0 | 0 | 0 | 0 | 2 | 3 | 3 | 5 | 8 | 8 | 7 | 6 | 5 | 4 | 7 |
| K | 0 | 0 | 0 | 0 | 1 | 4 | 3 | 4 | 7 | 10 | 9 | 8 | 7 | 6 | 8 |
| C | 0 | 0 | 0 | 0 | 0 | 3 | 3 | 3 | 6 | 9 | 9 | 8 | 7 | 6 | 9 |
| W | 0 | 0 | 0 | 0 | 0 | 2 | 2 | 2 | 5 | 8 | 8 | 11 | 10 | 9 | 10 |
| D | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 4 | 7 | 7 | 10 | 10 | 9 | 11 |
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | |

**Figure 37.6** Traceback with resulting local alignment

Now that we have the basic Smith-Waterman local alignment algorithm, we're going to add three features: substitution matrices, affine gap penalties, and scoring statistics.

**Substitution matrices**. So far, the scoring function gives a fixed score of $\alpha$ to every matching position and a fixed score of $\beta$ to every mismatched position, no matter what the amino acids are. However, mutations often substitute one amino acid for another. Sometimes the mutated protein's shape, and thus its function, are similar to the unmutated protein's shape and function. Therefore, penalizing each mismatch the same, or rewarding each match the same, regardless of the amino acids involved, does not accurately reflect the proteins' biological properties. Instead, $\delta(a_i, b_j)$ should yield different scores for different pairs of amino acids $(a_i, b_j)$.

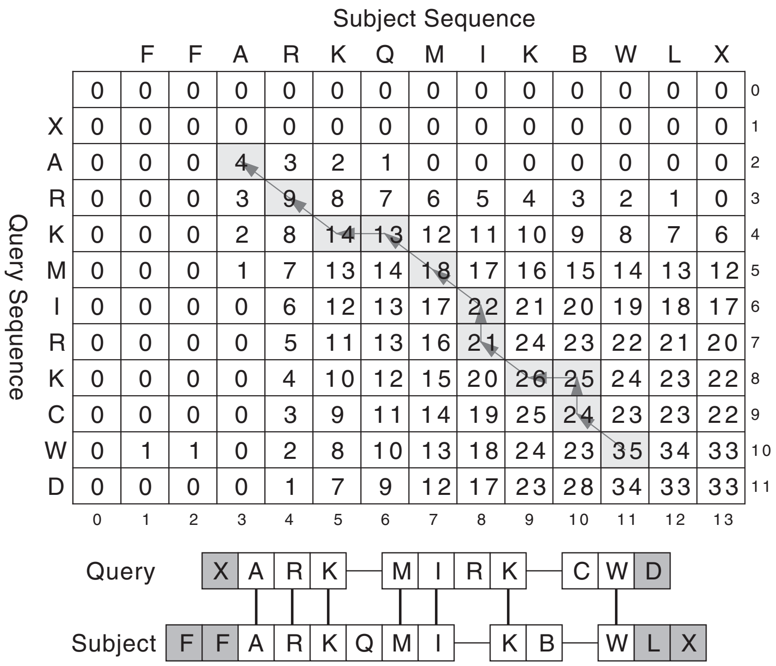
A **substitution matrix** is a table listing the alignment score when query letter $a_i$ is aligned with subject letter $b_j$ for every possible pair $(a_i, b_j)$. Various protein substitution matrices are commonly used, including the Point Accepted Mutation (PAM) family of matrices and the Block Substitution Matrix (BLOSUM) family of matrices. We will use the BLOSUM-62 matrix (Table 37.1). This is the default substitution matrix in the Basic Local Alignment Search Tool (BLAST), the popular protein sequence database-querying program.

**Table 37.1** The BLOSUM-62 substitution matrix

| | A | R | N | D | C | Q | E | G | H | I | L | K | M | F | P | S | T | W | Y | V | B | Z | X | * |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| A | 4 | -1 | -2 | -2 | 0 | -1 | -1 | 0 | -2 | -1 | -1 | -1 | -1 | -2 | -1 | 1 | 0 | -3 | -2 | 0 | -2 | -1 | 0 | -4 |
| R | -1 | 5 | 0 | -2 | -3 | 1 | 0 | -2 | 0 | -3 | -2 | 2 | -1 | -3 | -2 | -1 | -1 | -3 | -2 | -3 | -1 | 0 | -1 | -4 |
| N | -2 | 0 | 6 | 1 | -3 | 0 | 0 | 0 | 1 | -3 | -3 | 0 | -2 | -3 | -2 | 1 | 0 | -4 | -2 | -3 | 3 | 0 | -1 | -4 |
| D | -2 | -2 | 1 | 6 | -3 | 0 | 2 | -1 | -1 | -3 | -4 | -1 | -3 | -3 | -1 | 0 | -1 | -4 | -3 | -3 | 4 | 1 | -1 | -4 |
| C | 0 | -3 | -3 | -3 | 9 | -3 | -4 | -3 | -3 | -1 | -1 | -3 | -1 | -2 | -3 | -1 | -1 | -2 | -2 | -1 | -3 | -3 | -2 | -4 |
| Q | -1 | 1 | 0 | 0 | -3 | 5 | 2 | -2 | 0 | -3 | -2 | 1 | 0 | -3 | -1 | 0 | -1 | -2 | -1 | -2 | 0 | 3 | -1 | -4 |
| E | -1 | 0 | 0 | 2 | -4 | 2 | 5 | -2 | 0 | -3 | -3 | 1 | -2 | -3 | -1 | 0 | -1 | -3 | -2 | -2 | 1 | 4 | -1 | -4 |
| G | 0 | -2 | 0 | -1 | -3 | -2 | -2 | 6 | -2 | -4 | -4 | -2 | -3 | -3 | -2 | 0 | -2 | -2 | -3 | -3 | -1 | -2 | -1 | -4 |
| H | -2 | 0 | 1 | -1 | -3 | 0 | 0 | -2 | 8 | -3 | -3 | -1 | -2 | -1 | -2 | -1 | -2 | -2 | 2 | -3 | 0 | 0 | -1 | -4 |
| I | -1 | -3 | -3 | -3 | -1 | -3 | -3 | -4 | -3 | 4 | 2 | -3 | 1 | 0 | -3 | -2 | -1 | -3 | -1 | 3 | -3 | -3 | -1 | -4 |
| L | -1 | -2 | -3 | -4 | -1 | -2 | -3 | -4 | -3 | 2 | 4 | -2 | 2 | 0 | -3 | -2 | -1 | -2 | -1 | 1 | -4 | -3 | -1 | -4 |
| K | -1 | 2 | 0 | -1 | -3 | 1 | 1 | -2 | -1 | -3 | -2 | 5 | -1 | -3 | -1 | 0 | -1 | -3 | -2 | -2 | 0 | 1 | -1 | -4 |
| M | -1 | -1 | -2 | -3 | -1 | 0 | -2 | -3 | -2 | 1 | 2 | -1 | 5 | 0 | -2 | -1 | -1 | -1 | -1 | 1 | -3 | -1 | -1 | -4 |
| F | -2 | -3 | -3 | -3 | -2 | -3 | -3 | -3 | -1 | 0 | 0 | -3 | 0 | 6 | -4 | -2 | -2 | 1 | 3 | -1 | -3 | -3 | -1 | -4 |
| P | -1 | -2 | -2 | -1 | -3 | -1 | -1 | -2 | -2 | -3 | -3 | -1 | -2 | -4 | 7 | -1 | -1 | -4 | -3 | -2 | -2 | -1 | -2 | -4 |
| S | 1 | -1 | 1 | 0 | -1 | 0 | 0 | 0 | -1 | -2 | -2 | 0 | -1 | -2 | -1 | 4 | 1 | -3 | -2 | -2 | 0 | 0 | 0 | -4 |
| T | 0 | -1 | 0 | -1 | -1 | -1 | -1 | -2 | -2 | -1 | -1 | -1 | -1 | -2 | -1 | 1 | 5 | -2 | -2 | 0 | -1 | -1 | 0 | -4 |
| W | -3 | -3 | -4 | -4 | -2 | -2 | -3 | -2 | -2 | -3 | -2 | -3 | -1 | 1 | -4 | -3 | -2 | 11 | 2 | -3 | -4 | -3 | -2 | -4 |
| Y | -2 | -2 | -2 | -3 | -2 | -1 | -2 | -3 | 2 | -1 | -1 | -2 | -1 | 3 | -3 | -2 | -2 | 2 | 7 | -1 | -3 | -2 | -1 | -4 |
| V | 0 | -3 | -3 | -3 | -1 | -2 | -2 | -3 | -3 | 3 | 1 | -2 | 1 | -1 | -2 | -2 | 0 | -3 | -1 | 4 | -3 | -2 | -1 | -4 |
| B | -2 | -1 | 3 | 4 | -3 | 0 | 1 | -1 | 0 | -3 | -4 | 0 | -3 | -3 | -2 | 0 | -1 | -4 | -3 | -3 | 4 | 1 | -1 | -4 |
| Z | -1 | 0 | 0 | 1 | -3 | 3 | 4 | -2 | 0 | -3 | -3 | 1 | -1 | -3 | -1 | 0 | -1 | -3 | -2 | -2 | 1 | 4 | -1 | -4 |
| X | 0 | -1 | -1 | -1 | -2 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -2 | 0 | 0 | -2 | -1 | -1 | -1 | -1 | -1 | -4 |
| * | -4 | -4 | -4 | -4 | -4 | -4 | -4 | -4 | -4 | -4 | -4 | -4 | -4 | -4 | -4 | -4 | -4 | -4 | -4 | -4 | -4 | -4 | -4 | 1 |

ftp://ftp.ncbi.nih.gov/blast/matrices/BLOSUM62

Figure 37.7 shows the scoring matrix for the two example sequences when the BLOSUM-62 substitution matrix is used. The gap penalty is still $\gamma = -1$. This time, the best alignment score is +35. The resulting local alignment is slightly different from Figure 37.6.



| | | F | F | A | R | K | Q | M | I | K | B | W | L | X | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| X | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| A | 0 | 0 | 0 | 4 | 3 | 2 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2 |
| R | 0 | 0 | 0 | 3 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 3 |
| K | 0 | 0 | 0 | 2 | 8 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 4 |
| M | 0 | 0 | 0 | 1 | 7 | 13 | 14 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 5 |
| I | 0 | 0 | 0 | 0 | 6 | 12 | 13 | 17 | 22 | 21 | 20 | 19 | 18 | 17 | 6 |
| R | 0 | 0 | 0 | 0 | 5 | 11 | 13 | 16 | 21 | 24 | 23 | 22 | 21 | 20 | 7 |
| K | 0 | 0 | 0 | 0 | 4 | 10 | 12 | 15 | 20 | 26 | 25 | 24 | 23 | 22 | 8 |
| C | 0 | 0 | 0 | 0 | 3 | 9 | 11 | 14 | 19 | 25 | 24 | 23 | 23 | 22 | 9 |
| W | 0 | 1 | 1 | 0 | 2 | 8 | 10 | 13 | 18 | 24 | 23 | 35 | 34 | 33 | 10 |
| D | 0 | 0 | 0 | 0 | 1 | 7 | 9 | 12 | 17 | 23 | 28 | 34 | 33 | 33 | 11 |
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | |

**Figure 37.7** Local alignment using BLOSUM-62

**Affine gap penalties**. A single mutation sometimes inserts or deletes multiple consecutive amino acids in a protein. If the mutated protein sequence is aligned with the original protein sequence, the mutation shows up as a multiple-position gap. So far, the scoring function deducts a fixed penalty $\gamma$ for each gap position, so that the penalty is proportional to the gap length. But if a single mutation can result in a multiple-position gap, this is too harsh a penalty. A more realistic gap scoring function is

$$\gamma_{\text{open}} + \gamma_{\text{ext}} \cdot L \tag{37.1}$$

where $\gamma_{\text{open}} < 0$ is the **gap opening penalty** (the score for starting, or "opening," a gap), $\gamma_{\text{ext}} < 0$ is the **gap extension penalty** (the score for extending a gap), and $L$ is the length of the gap (the number of gap positions). $\gamma_{\text{ext}}$ is usually considerably less than $\gamma_{\text{open}}$. Because (37.1) is an affine function of the gap length, this gap scoring function is called an **affine gap penalty**.

Rewriting (37.1) slightly gives the gap scoring function,

$$\gamma_{\text{exist}} + \gamma_{\text{ext}} \cdot (L-1) \tag{37.2}$$

where $\gamma_{\text{exist}} = \gamma_{\text{open}} + \gamma_{\text{ext}}$ is the **gap existence penalty**. $\gamma_{\text{exist}}$ is the score for the first position of a gap; $\gamma_{\text{ext}}$ is the score for the second and subsequent positions of a gap. The BLAST program uses $\gamma_{\text{exist}} = -11$ and $\gamma_{\text{ext}} = -1$ by default.

To incorporate affine gap penalties into the Smith-Waterman algorithm, we must use two additional scoring matrices. $GA[i, j]$ is the alignment score if query letter $a_i$ is aligned with a gap at position $j$ in subject sequence $B$. We have two choices for extending $GA$:

- Query letter $a_{i-1}$ is not aligned with a gap in $B$, and we start a gap. If so, $GA[i, j] = S[i-1, j] + \gamma_{\text{exist}}$.
- Query letter $a_{i-1}$ is aligned with a gap in $B$, and we extend the gap. If so, $GA[i, j] = GA[i-1, j] + \gamma_{\text{ext}}$.

We make the choice that results in the highest value for $GA[i, j]$. Likewise, $GB[i, j]$ is the alignment score if subject letter $b_j$ is aligned with a gap at position $i$ in query sequence $A$. We have two choices for extending $GB$:

- Subject letter $b_{j-1}$ is not aligned with a gap in $A$, and we start a gap. If so, $GB[i, j] = S[i, j-1] + \gamma_{\text{exist}}$.
- Subject letter $b_{j-1}$ is aligned with a gap in $A$, and we extend the gap. If so, $GB[i, j] = GB[i, j-1] + \gamma_{\text{ext}}$.

We make the choice that results in the highest value for $GB[i, j]$. We then use $GA[i, j]$ and $GB[i, j]$ to choose the value for $S[i, j]$. The pseudocode follows:

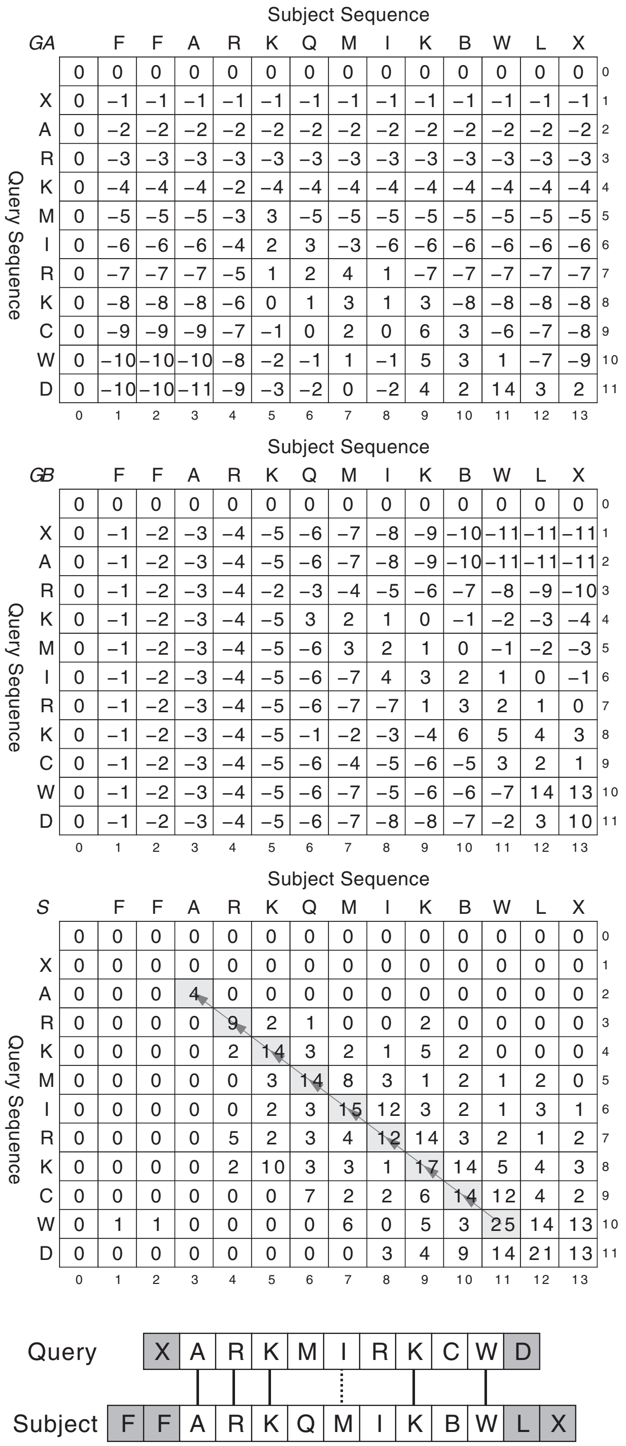
```
$A[1..M] \leftarrow$ Query sequence
$B[1..N] \leftarrow$ Subject sequence
Create scoring matrices $S[0..M, 0..N]$, $GA[0..M, 0..N]$, $GB[0..M, 0..N]$
(All elements in row 0 of $S$, $GA$, $GB$) $\leftarrow 0$
(All elements in column 0 of $S$, $GA$, $GB$) $\leftarrow 0$
For $i = 1$ to $M$:
    For $j = 1$ to $N$:
            $GA[i, j] \leftarrow \max(S[i-1, j] + \gamma_{\text{exist}}, GA[i-1, j] + \gamma_{\text{ext}})$
            $GB[i, j] \leftarrow \max(S[i, j-1] + \gamma_{\text{exist}}, GB[i, j-1] + \gamma_{\text{ext}})$
            $S[i, j] \leftarrow \max(S[i-1, j-1] + \delta(a_i, b_j), GA[i, j], GB[i, j], 0)$
```

We must also modify the traceback. We start with the largest entry in the $S$ matrix, as before. But as we follow the $S$ matrix backward, if we decide that query letter $a_i$ is aligned with a gap in $B$, we switch to following the $GA$ matrix; when we decide that query letter $a_i$ starts a gap, we switch back to the $S$ matrix. Likewise, if we decide that subject letter $b_j$ is aligned with a gap in $A$, we switch to following the $GB$ matrix; when we decide that subject letter $b_j$ starts a gap, we switch back to the $S$ matrix. The pseudocode follows:

```
$(i, j) \leftarrow$ (row, column) of largest entry in $S$
Follow $S$ matrix
While $S[i, j] \neq 0$:
    If following $S$ matrix:
            If $S[i, j] = S[i-1, j-1] + \delta(a_i, b_j)$:
                    Query letter $a_i$ is aligned with subject letter $b_j$
                    $(i, j) \leftarrow (i-1, j-1)$
```

```
        Else if $S[i, j] = GA[i, j]$:
                Follow $GA$ matrix
        Else:
                Follow $GB$ matrix
Else if following $GA$ matrix:
        Query letter $a_i$ is aligned with a gap in $B$
        If $GA[i, j] = S[i–1, j] + \gamma_{\text{exist}}$:
                Follow $S$ matrix
        $(i, j) \leftarrow (i–1, j)$
Else:
        Subject letter $b_j$ is aligned with a gap in $A$
        If $GB[i, j] = S[i, j–1] + \gamma_{\text{exist}}$:
                Follow $S$ matrix
        $(i, j) \leftarrow (i, j–1)$
```

Figure 37.8 shows the scoring matrix for the two example sequences when the BLOSUM-62 substitution matrix is used along with the affine gap penalty parameters ($\gamma_{\text{exist}}$, $\gamma_{\text{ext}}$) = (–11, –1). This time, due to the large gap existence penalty, the highest-scoring alignment has no gaps. The alignment also illustrates a position with mismatched amino acids, I and M, that nonetheless contributes a positive score (+1) according to the BLOSUM-62 substitution matrix.



Subject Sequence

Query Sequence

| GA | | F | F | A | R | K | Q | M | I | K | B | W | L | X | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| X | 0 | –1 | –1 | –1 | –1 | –1 | –1 | –1 | –1 | –1 | –1 | –1 | –1 | –1 | 1 |
| A | 0 | –2 | –2 | –2 | –2 | –2 | –2 | –2 | –2 | –2 | –2 | –2 | –2 | –2 | 2 |
| R | 0 | –3 | –3 | –3 | –3 | –3 | –3 | –3 | –3 | –3 | –3 | –3 | –3 | –3 | 3 |
| K | 0 | –4 | –4 | –4 | –2 | –4 | –4 | –4 | –4 | –4 | –4 | –4 | –4 | –4 | 4 |
| M | 0 | –5 | –5 | –5 | –3 | 3 | –5 | –5 | –5 | –5 | –5 | –5 | –5 | –5 | 5 |
| I | 0 | –6 | –6 | –6 | –4 | 2 | 3 | –3 | –6 | –6 | –6 | –6 | –6 | –6 | 6 |
| R | 0 | –7 | –7 | –7 | –5 | 1 | 2 | 4 | 1 | –7 | –7 | –7 | –7 | –7 | 7 |
| K | 0 | –8 | –8 | –8 | –6 | 0 | 1 | 3 | 1 | 3 | –8 | –8 | –8 | –8 | 8 |
| C | 0 | –9 | –9 | –9 | –7 | –1 | 0 | 2 | 0 | 6 | 3 | –6 | –7 | –8 | 9 |
| W | 0 | –10 | –10 | –10 | –8 | –2 | –1 | 1 | –1 | 5 | 3 | 1 | –7 | –9 | 10 |
| D | 0 | –10 | –10 | –11 | –9 | –3 | –2 | 0 | –2 | 4 | 2 | 14 | 3 | 2 | 11 |
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | |

Subject Sequence

Query Sequence

| GB | | F | F | A | R | K | Q | M | I | K | B | W | L | X | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| X | 0 | –1 | –2 | –3 | –4 | –5 | –6 | –7 | –8 | –9 | –10 | –11 | –11 | –11 | 1 |
| A | 0 | –1 | –2 | –3 | –4 | –5 | –6 | –7 | –8 | –9 | –10 | –11 | –11 | –11 | 2 |
| R | 0 | –1 | –2 | –3 | –4 | –2 | –3 | –4 | –5 | –6 | –7 | –8 | –9 | –10 | 3 |
| K | 0 | –1 | –2 | –3 | –4 | –5 | 3 | 2 | 1 | 0 | –1 | –2 | –3 | –4 | 4 |
| M | 0 | –1 | –2 | –3 | –4 | –5 | –6 | 3 | 2 | 1 | 0 | –1 | –2 | –3 | 5 |
| I | 0 | –1 | –2 | –3 | –4 | –5 | –6 | –7 | 4 | 3 | 2 | 1 | 0 | –1 | 6 |
| R | 0 | –1 | –2 | –3 | –4 | –5 | –6 | –7 | –7 | 1 | 3 | 2 | 1 | 0 | 7 |
| K | 0 | –1 | –2 | –3 | –4 | –5 | –1 | –2 | –3 | –4 | 6 | 5 | 4 | 3 | 8 |
| C | 0 | –1 | –2 | –3 | –4 | –5 | –6 | –4 | –5 | –6 | –5 | 3 | 2 | 1 | 9 |
| W | 0 | –1 | –2 | –3 | –4 | –5 | –6 | –7 | –5 | –6 | –6 | –7 | 14 | 13 | 10 |
| D | 0 | –1 | –2 | –3 | –4 | –5 | –6 | –7 | –8 | –8 | –7 | –2 | 3 | 10 | 11 |
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | |

Subject Sequence

Query Sequence

| S | | F | F | A | R | K | Q | M | I | K | B | W | L | X | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| X | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| A | 0 | 0 | 0 | 4 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2 |
| R | 0 | 0 | 0 | 0 | 9 | 2 | 1 | 0 | 0 | 2 | 0 | 0 | 0 | 0 | 3 |
| K | 0 | 0 | 0 | 0 | 2 | 14 | 3 | 2 | 1 | 5 | 2 | 0 | 0 | 0 | 4 |
| M | 0 | 0 | 0 | 0 | 0 | 3 | 14 | 8 | 3 | 1 | 2 | 1 | 2 | 0 | 5 |
| I | 0 | 0 | 0 | 0 | 0 | 2 | 3 | 15 | 12 | 3 | 2 | 1 | 3 | 1 | 6 |
| R | 0 | 0 | 0 | 0 | 5 | 2 | 3 | 4 | 12 | 14 | 3 | 2 | 1 | 2 | 7 |
| K | 0 | 0 | 0 | 0 | 2 | 10 | 3 | 3 | 1 | 17 | 14 | 5 | 4 | 3 | 8 |
| C | 0 | 0 | 0 | 0 | 0 | 0 | 7 | 2 | 2 | 6 | 14 | 12 | 4 | 2 | 9 |
| W | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 6 | 0 | 5 | 3 | 25 | 14 | 13 | 10 |
| D | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 3 | 4 | 9 | 14 | 21 | 13 | 11 |
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | |

Query X A R K M I R K C W D
Subject F F A R K Q M I K B W L X

**Figure 37.8** Local alignment using BLOSUM-62 and affine gap penalties

**Scoring statistics**. If we tried aligning a protein's query sequence against the subject sequences in a large protein sequence database, we'd expect to see a large alignment score when the query protein was closely related to the database protein. However, we'd also expect to see a small but nonzero alignment score where a piece of the query sequence just happened to match a piece of the subject sequence by chance, although the proteins in question had nothing to do with each other. How can we tell which is which? How large does the score have to be for the alignment to be biologically relevant?

Protein database-querying programs like BLAST use the ***E*-value** to answer this question. Given a certain query sequence, a certain database of subject sequences, and a certain alignment score *S*, the *E*-value is the expected number of alignments with a score of *S* or greater that will be encountered purely by chance. The smaller the *E*-value of an alignment's score, the more statistically significant, and, therefore, the more biologically relevant the alignment is.

The formula for the *E*-value is

$$E = K\ M\ N\ e^{-\lambda S} \tag{37.3}$$

where *M* is the query sequence length, *N* is the sum of the subject sequences' lengths in the database, and *S* is the alignment score. Note that as *S* increases, *E* decreases (and becomes more statistically significant). *K* and λ are parameters that depend on the alignment algorithm. For local alignments using the BLOSUM-62 substitution matrix along with the affine gap penalties ($\gamma_{\text{exist}}$, $\gamma_{\text{ext}}$) = (–11, –1), the parameters have been determined empirically to be $K = 0.035$ and $\lambda = 0.252$.

Protein database querying programs like BLAST also report the alignment score as a **bit score**. The formula for the bit score is the following:

$$bitS = \frac{\lambda S - 1\text{n } K}{1\text{n } 2} \tag{37.4}$$

In terms of the bit score, the *E*-value is the following:

$$E = M\ N\ 2^{-bitS} \tag{37.5}$$

Formula (37.4) normalizes the algorithm-dependent raw alignment score to an algorithm-independent bit score. Formula (37.5) for the *E*-value then holds true for any alignment algorithm. Thus, an alignment with a higher bit score is better than an alignment with a lower bit score, even if different algorithms were used to produce the alignments.

## 37.3 A Protein Sequence Query Example

Before starting our software design, let's look at an example of a protein database query. To do that, we need a protein sequence database. The Universal Protein Resource (UniProt), a collaboration between the European Bioinformatics Institute in Cambridge, UK, the Swiss Institute of Bioinformatics, and the Protein Information Resource at Georgetown University, provides several Web-accessible protein sequence databases. We'll use the Swiss-Prot database. You can download a file containing the sequences of all the proteins in the database. Here are a few sequences from the file:

```
>Q43495|108_SOLLC Protein 108 precursor - Solanum lycopersicum (Tomato) (Lyc
MASVKSSSSSSSSSFISLLLLILLVIVLQSQVIECQPQQSCTASLTGLNVCAPFLVPGSP
TASTECCNAVQSINHDCMCNTMRIAAQIPAQCNLPPLSCSAN
>P18646|10KD_VIGUN 10 kDa protein precursor - Vigna unguiculata (Cowpea)
MEKKSIAGLCFLFLVLFVAQEVVVQSEAKTCENLVDTYRGPCFTTGSCDDHCKNKEHLLS
GRCRDDVRCWCTRNC
>P13813|110KD_PLAKN 110 kDa antigen - Plasmodium knowlesi
FNSNMLRGSVCEEDVSLMTSIDNMIEEIDFYEKEIYKGSHSGGVIKGMDYDLEDDENDED
EMTEQMVEEVADHITQDMIDEVAHHVLDNITHDMAHMEEIVHGLSGDVTQIKEIVQKVNV
AVEKVKHIVETEETQKTVEPEQIEETQNTVEPEQTEETQKTVEPEQTEETQNTVEPEQIE
ETQKTVEPEQTEEAQKTVEPEQTEETQKTVEPEQTEETQKTVEPEQTEETQKTVEPEQTE
ETQKTVEPEQTEETQKTVEPEQTEETQKTVEPEQTEETQNTVEPEPTQETQNTVEP
>Q9XHP0|11S2_SESIN 11S globulin seed storage protein 2 precursor – Sesamum i
MVAFKFLLALSLSLLVSAAIAQTREPRLTQGQQCRFQRISGAQPSLRIQSEGGTTELWDE
RQEQFQCAGIVAMRSTIRPNGLSLPNYHPSPRLVYIERGQGLISIMVPGCAETYQVHRSQ
RTMERTEASEQQDRGSVRDLHQKVHRLRQGDIVAIPSGAAHWCYNDGSEDLVAVSINDVN
HLSNQLDQKFRAFYLAGGVPRSGEQEQQARQTFHNIFRAFDAELLSEAFNVPQETIRRMQ
SEEEERGLIVMARERMTFVRPDEEEGEQEHRGRQLDNGLEETFCTMKFRTNVESRREADI
FSRQAGRVHVVDRNKLPILKYMDLSAEKGNLYSNALVSPDWSMTGHTIVYVTRGDAQVQV
VDHNGQALMNDRVNQGEMFVVPQYYTSTARAGNNGFEWVAFKTTGSPMRSPLAGYTSVIR
AMPLQVITNSYQISPNQAQALKMNRGSQSFLLSPGGRRS
```

The database is a plain text file in **FASTA format**. FASTA is a protein sequence analysis program developed by David Lipman and William Pearson, originally published in 1985; the format in which the FASTA program stores sequences has become the de facto standard. A line beginning with a '>' character marks the start of a protein. This first line is the protein's description. By convention in the Swiss-Prot database, the description begins with the protein's "accession number," or ID. You can search for the accession number on the UniProt Web site and find full details about the protein. The lines following the description contain the protein's amino acid sequence. (Line breaks are not significant.)

To query the database, we must supply a file with a query sequence in FASTA format. Here is the query sequence for human insulin:

```
>Human insulin (B chain 1..30, A chain 31..51) - Homo sapiens (Human)
FVNQHLCGSHLVEALYLVCGERGFFYTPKT
GIVEQCCTSICSLYQLENYCN
```

Here is what the protein database-querying program from later in the chapter printed when told to run the preceding query against the Swiss-Prot database. The printout begins with the query sequence description.

```
Query Description:
>Human insulin (B chain 1..30, A chain 31..51) - Homo sapiens (Human)
Length = 51
```

The program ran the Smith-Waterman algorithm for the query sequence and each database sequence, computing the local alignments, the bit scores, and the *E*-values. The next part of the printout lists the descriptions of the proteins in the database that matched the query. A match is defined to be a local alignment with an *E*-value less than or equal to 10. (The *E*-value threshold can be specified on the command line.) The matching proteins are listed in ascending order of *E*-value, so that the most biologically relevant matches appear first. The program found 143 matches.

```
                                                                Bit  E-
Subject Description                                           Score  Value
>P67973|INS_BALPH Insulin [Contains: Insulin B chain; Ins...    108  2e-23
>P01316|INS_ELEMA Insulin [Contains: Insulin B chain; Ins...    108  2e-23
>P67974|INS_PHYCA Insulin [Contains: Insulin B chain; Ins...    108  2e-23
>P01324|INS_ACOCA Insulin [Contains: Insulin B chain; Ins...    106  7e-23
>P01314|INS_BALBO Insulin [Contains: Insulin B chain; Ins...    104  3e-22
. . .
>P26726|BXA3_BOMMO Bombyxin A-3 precursor - Bombyx mori (...   30.3  5.4
>Q17192|BXA1_BOMMO Bombyxin A-1 precursor - Bombyx mori (...   29.9  6.9
>P26727|BXA4_BOMMO Bombyxin A-4 precursor - Bombyx mori (...   29.9  6.9
>P26729|BXA6_BOMMO Bombyxin A-6 precursor - Bombyx mori (...   29.9  6.9
>P26730|BXA7_BOMMO Bombyxin A-7 precursor - Bombyx mori (...   29.9  6.9
```

Next, the printout gives the details of each alignment, again in order from lowest *E*-value to highest. Here is the first.

```
>P67973|INS_BALPH Insulin [Contains: Insulin B chain; Insulin A chain] -
Balaenoptera physalus (Finback whale) (Common rorqual)
Length = 51

Score = 108 bits (284), Expect = 2e-23
Identities = 50/51 (98%), Positives = 50/51 (98%), Gaps = 0/51 (0%)

Query     1  FVNQHLCGSHLVEALYLVCGERGFFYTPKTGIVEQCCTSICSLYQLENYCN    51
             FVNQHLCGSHLVEALYLVCGERGFFYTPK GIVEQCCTSICSLYQLENYCN
Sbjct     1  FVNQHLCGSHLVEALYLVCGERGFFYTPKAGIVEQCCTSICSLYQLENYCN    51
```

The line labeled "Query" gives the starting and ending indexes (1 and 51) of the query sequence in the alignment, along with the actual amino acids. The line labeled "Sbjct" does the same for the subject sequence. The line in between shows matches and mismatches. A matching position echoes the amino acid letter; a space denotes a mismatched position. Finback whale insulin is the same as human insulin, except for one amino acid.

Here is the second alignment.

```
>P01316|INS_ELEMA Insulin [Contains: Insulin B chain; Insulin A chain] -
Elephas maximus (Indian elephant)
Length = 51

Score = 108 bits (284), Expect = 2e-23
Identities = 49/51 (96%), Positives = 50/51 (98%), Gaps = 0/51 (0%)
```

```
Query      1   FVNQHLCGSHLVEALYLVCGERGFFYTPKTGIVEQCCTSICSLYQLENYCN    51
               FVNQHLCGSHLVEALYLVCGERGFFYTPKTGIVEQCCT +CSLYQLENYCN
Sbjct      1   FVNQHLCGSHLVEALYLVCGERGFFYTPKTGIVEQCCTGVCSLYQLENYCN    51
```

Indian elephant insulin is the same as human insulin, except for two amino acids. For one of the mismatched positions, though, the BLOSUM-62 substitution matrix yielded a positive score. This is denoted by a "+" sign.

You may be wondering why human insulin, which ought to be an exact match for the query sequence, does not show up first in the list. Here is the alignment for human insulin; it shows up as the 21st:

```
>P01308|INS_HUMAN Insulin precursor [Contains: Insulin B chain; Insulin A
chain] - Homo sapiens (Human)
Length = 110

Score = 93.5 bits (244), Expect = 5e-19
Identities = 51/86 (59%), Positives = 51/86 (59%), Gaps = 35/86 (41%)

Query   1  FVNQHLCGSHLVEALYLVCGERGFFYTPKT------------------------------   30
           FVNQHLCGSHLVEALYLVCGERGFFYTPKT
Sbjct  25  FVNQHLCGSHLVEALYLVCGERGFFYTPKTRREAEDLQVGQVELGGGPGAGSLQPLALEG   84

Query  31  -----GIVEQCCTSICSLYQLENYCN   51
                GIVEQCCTSICSLYQLENYCN
Sbjct  85  SLQKRGIVEQCCTSICSLYQLENYCN  110
```

Note the large gap in the alignment from positions 55–89 in the subject sequence. Also note that positions 1–24 in the subject sequence are not part of the alignment. This is because what's in the database is the human insulin *precursor* protein, not human insulin itself. To synthesize insulin, the body first assembles the precursor protein as a single 110-amino-acid chain. Special enzymes cleave the precursor protein into the B chain (positions 25–54) and the A chain (positions 90–110). The A and B chains are attached by two sulfur atoms, each sulfur atom joining a certain position on the A chain with a certain position on the B chain (disulfide bonds). There is also a third disulfide bond between two positions on the A chain. The overall protein's shape is determined by the individual chains' shapes and the disulfide bonds' locations.

When the query sequence for human insulin is aligned with the subject sequence for the human insulin precursor protein, the large gap between the A and B chains reduces the alignment score compared to a subject sequence that is just insulin. Disregarding the gap, the query sequence is identical to the subject sequence.

As fodder for your next trivia party, you may be interested in what the printout reveals about the degree of similarity (number of differing amino acids) between human insulin and other animals' insulin:

- Identical: green monkey, gorilla, crab-eating macaque, chimpanzee, orangutan
- One amino acid different: finback whale, sperm whale, pig, rabbit, ground squirrel, dog
- Two amino acids different: Indian elephant, hamster, horse, fat sand rat, Norway rat

- Three amino acids different: Egyptian spiny mouse, pollack whale, cow, mouse
- Four amino acids different: dromedary, goat, opossum, sheep, cat

Based on their insulin, humans have a greater affinity to rats than to cats.

## 37.4 Sequential Program

Figure 37.9 shows the classes from which the sequential protein sequence database-querying program is built and their "uses" relationships. (A→B means class A uses class B.) These classes are in package edu.rit.compbio.seq. The main program is class FindProteinSeq. We'll describe what every class does, but study the code for only a few key classes.

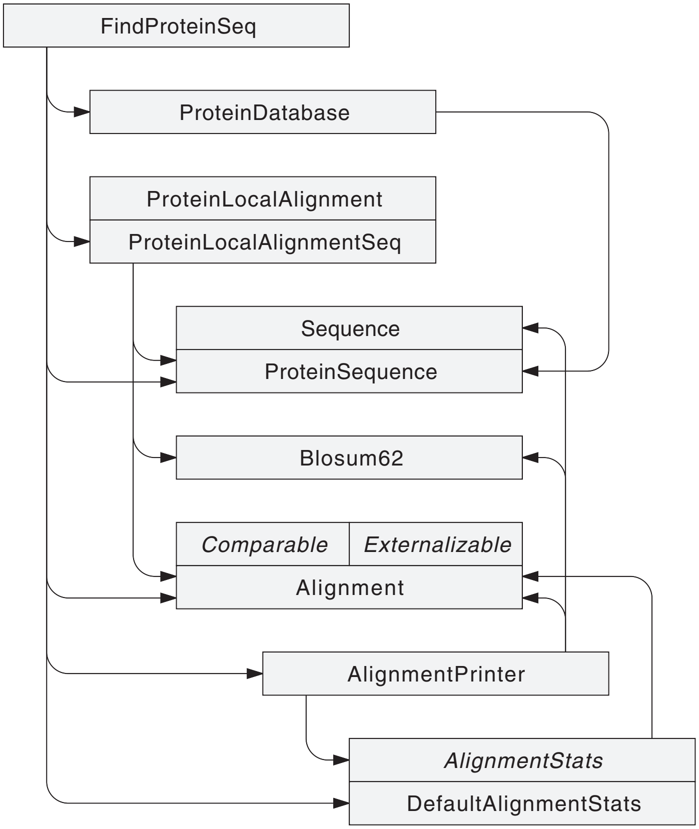


**Figure 37.9** FindProteinSeq class relationships

- Class Sequence is the abstract base class for a biological sequence, including the description, the characters in the sequence, and a sequence ID.
- Class ProteinSequence, a subclass of class Sequence, provides a protein sequence. Constructors are provided to read a protein sequence from a FASTA format file

or from a string. (Another subclass of class Sequence, not presently implemented, could provide the other kind of biological sequence, a DNA sequence.)

- Class ProteinDatabase is used to read protein sequences from a protein database file in FASTA format. An index file that gives the offset of each protein in the database file is associated with the database file. Given a protein index, the ProteinDatabase object reads the offset at that index from the index file, reads the protein sequence at that offset from the database file, and returns a ProteinSequence object. This allows the program to quickly read an arbitrary protein sequence from the database. Class ProteinDatabase also includes a main program that creates the index file, given the database file.
- Class Blosum62 encapsulates the BLOSUM-62 substitution matrix.
- Class Alignment stores the result of an alignment algorithm between a query sequence and a subject sequence. It includes the database indexes of both sequences, the starting and ending positions of the alignment in both sequences, the state of each alignment position (query letter aligned with subject letter, query letter aligned with gap, or subject letter aligned with gap), and the alignment's raw score. Class Alignment implements interface java.lang.Comparable so that alignment objects can be sorted into descending order of alignment score. Class Alignment also implements interface java.io.Externalizable so that alignment objects can be serialized and sent in messages between the processes of a parallel program.
- Class AlignmentPrinter has a method to print a summary of an alignment (description, bit score, *E*-value) and a method to print the details of an alignment.
- Interface AlignmentStats specifies methods to calculate an alignment's bit score and *E*-value.
- Class DefaultAlignmentStats implements interface AlignmentStats with the formulas and parameters we are using for the bit score (37.4) and *E*-value (37.3). (A program that used a different alignment algorithm would have to implement interface AlignmentStats differently.)
- Class ProteinLocalAlignment is the abstract base class for the Smith-Waterman local alignment algorithm using a substitution matrix and affine gap penalties. The base class has fields and setter methods for several alignment parameters, an abstract `align()` method to be implemented in a subclass, and a private method for doing the traceback.

```
package edu.rit.compbio.seq;
import java.io.ByteArrayOutputStream;
public abstract class ProteinLocalAlignment
   {
   // Substitution matrix.
   int[][] delta = Blosum62.matrix;
```

```
// Gap existence and extension penalties.
int g = -11;
int h = -1;

// Query sequence, ID, length.
byte[] A;
long myQueryId;
int myQueryLength;

// Subject sequence, ID, length.
byte[] B;
long mySubjectId;
int mySubjectLength;

// Score matrix.
int[][] S;

// Gap score matrices.
int[][] GA;
int[][] GB;

// Extra padding to avert cache interference.
long p0, p1, p2, p3, p4, p5, p6, p7;
long p8, p9, pa, pb, pc, pd, pe, pf;

/**
 * Construct a new protein sequence local alignment object.
 */
public ProteinLocalAlignment()
   {
   }

/**
 * Set the protein substitution matrix. If not set, the default
 * is the BLOSUM-62 substitution matrix.
 */
public void setSubstitutionMatrix
   (int[][] matrix)
   {
   this.delta = matrix;
   }

/**
 * Set the gap existence penalty. If not set, the default is
 * -11.
 */
```

```
public void setGapExistencePenalty
   (int g)
   {
   this.g = g;
   }

/**
 * Set the gap extension penalty. If not set, the default is -1.
 */
public void setGapExtensionPenalty
   (int h)
   {
   this.h = h;
   }

/**
 * Set the query sequence.
 */
public void setQuerySequence
   (ProteinSequence theSequence,
    long theId)
   {
   A = theSequence.sequence();
   myQueryId = theId;
   myQueryLength = theSequence.length();
   int M = A.length;
   if (S == null || S.length < M+32) // Extra padding
      {
      S = new int [M+32] [];
      GA = new int [M+32] [];
      GB = new int [M+32] [];
      }
   }

/**
 * Set the subject sequence.
 */
public void setSubjectSequence
   (ProteinSequence theSequence,
    long theId)
   {
   if (A == null)
      {
      throw new IllegalStateException
         ("ProteinLocalAlignment.setSubjectSequence(): Query "+
          "sequence not set");
      }
   B = theSequence.sequence();
   mySubjectId = theId;
   mySubjectLength = theSequence.length();
   int N = B.length;
   if (S[0] == null || S[0].length < N+32) // Extra padding
      {
      int M = S.length-32;
      for (int i = 0; i < M; ++ i)
         {
         S[i] = new int [N+32];
         GA[i] = new int [N+32];
         GB[i] = new int [N+32];
         }
      }
   }

/**
 * Align the query sequence and the subject sequence.
 */
public abstract Alignment align()
   throws Exception;

/**
 * Compute the traceback and return the resulting alignment.
 */
Alignment computeTraceback
   (int theScore,
    int theQueryFinish,
    int theSubjectFinish)
   {
   // Set up alignment object.
   Alignment alignment = new Alignment();
   alignment.myQueryId = this.myQueryId;
   alignment.mySubjectId = this.mySubjectId;
   alignment.myQueryLength = this.myQueryLength;
   alignment.mySubjectLength = this.mySubjectLength;
   // Special case: No alignment found.
   if (theScore == 0)
      {
      alignment.myTraceback = new byte [0];
      return alignment;
      }

   // For recording alignment state at each position.
   ByteArrayOutputStream baos = new ByteArrayOutputStream();
```

```
// Trace backwards until we reach a score of 0.
int i = theQueryFinish;
int j = theSubjectFinish;
int theQueryStart = i;
int theSubjectStart = j;
int state = 0;
while (S[i][j] != 0)
   {
   switch (state)
      {
      case 0: // Tracing back through table S
         if (S[i][j] == S[i-1][j-1] + delta[A[i]][B[j]])
            {
            baos.write
               (Alignment.QUERY_ALIGNED_WITH_SUBJECT);
            theQueryStart = i;
            theSubjectStart = j;
            -- i;
            -- j;
            }
         else if (S[i][j] == GA[i][j])
            {
            state = 1;
            }
         else
            {
            state = 2;
            }
         break;
      case 1: // Tracing back through table GA
         baos.write (Alignment.QUERY_ALIGNED_WITH_GAP);
         if (GA[i][j] == S[i-1][j] + g) state = 0;
         theQueryStart = i;
         -- i;
         break;
      case 2: // Tracing back through table GB
         baos.write (Alignment.SUBJECT_ALIGNED_WITH_GAP);
         if (GB[i][j] == S[i][j-1] + g) state = 0;
         theSubjectStart = j;
         -- j;
         break;
      }
   }

// Record results.
alignment.myScore = theScore;
```

```
      alignment.myQueryStart = theQueryStart;
      alignment.mySubjectStart = theSubjectStart;
      alignment.myQueryFinish = theQueryFinish;
      alignment.mySubjectFinish = theSubjectFinish;
      alignment.myTraceback = baos.toByteArray();
      return alignment;
      }
   }
```

- Class ProteinLocalAlignmentSeq extends class ProteinLocalAlignment with a sequential implementation of the `align()` method. (Later, we will see another subclass with an SMP parallel implementation.)

```
package edu.rit.compbio.seq;
public class ProteinLocalAlignmentSeq
   extends ProteinLocalAlignment
   {
   /**
    * Construct a new protein sequence local alignment object.
    */
   public ProteinLocalAlignmentSeq()
      {
      super();
      }

   /**
    * Align the query sequence and the subject sequence.
    */
   public Alignment align()
      {
      // Verify preconditions.
      if (A == null)
         {
         throw new IllegalStateException
            ("ProteinLocalAlignmentSeq.align(): Query sequence "+
             "not set");
         }
      if (B == null)
         {
         throw new IllegalStateException
            ("ProteinLocalAlignmentSeq.align(): Subject sequence "+
             "not set");
         }
      int M = A.length - 1;
      int N = B.length - 1;
      // Do the Smith-Waterman algorithm in a single thread.
      int maxScore = 0;
      int theQueryFinish = 0;
      int theSubjectFinish = 0;
      for (int i = 1; i <= M; ++ i)
         {
         int A_i = A[i];
         int[] delta_A_i = delta[A_i];
         int[] S_im1 = S[i-1];
         int[] S_i = S[i];
         int[] GA_im1 = GA[i-1];
         int[] GA_i = GA[i];
         int[] GB_i = GB[i];
         int B_j, S_i_j, GA_i_j, GB_i_j;
         for (int j = 1; j <= N; ++ j)
            {
            B_j = B[j];
            GA_i_j = S_im1[j] + g;
            GA_i_j = Math.max (GA_i_j, GA_im1[j] + h);
            GB_i_j = S_i[j-1] + g;
            GB_i_j = Math.max (GB_i_j, GB_i[j-1] + h);
            S_i_j = S_im1[j-1] + delta_A_i[B_j];
            S_i_j = Math.max (S_i_j, GA_i_j);
            S_i_j = Math.max (S_i_j, GB_i_j);
            S_i_j = Math.max (S_i_j, 0);
            if (S_i_j > maxScore)
               {
               maxScore = S_i_j;
               theQueryFinish = i;
               theSubjectFinish = j;
               }
            S_i[j] = S_i_j;
            GA_i[j] = GA_i_j;
            GB_i[j] = GB_i_j;
            }
         }

      // Do the traceback.
      return computeTraceback
         (maxScore, theQueryFinish, theSubjectFinish);
      }
   }
```

Finally, class FindProteinSeq is the main program class for the sequential version of the protein sequence database-querying program. The command-line arguments are the following:

- Name of the file containing the query sequence in FASTA format.
- Name of the file containing the protein sequence database in FASTA format.
- Name of the database index file.
- *E*-value threshold for reporting a match. If omitted, the default is 10.

```
package edu.rit.compbio.seq;
import java.io.File;
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
public class FindProteinSeq
   {
   // Command line arguments.
   static File queryfile;
   static File databasefile;
   static File indexfile;
   static double expect;

   /**
    * Main program.
    */
   public static void main
      (String[] args)
      throws Exception
      {
      // Start timing.
      long t1 = System.currentTimeMillis();

      // Parse command line arguments.
      if (3 > args.length || args.length > 4) usage();
      queryfile = new File (args[0]);
      databasefile = new File (args[1]);
      indexfile = new File (args[2]);
      expect = 10.0;
      if (args.length == 4) expect = Double.parseDouble (args[3]);

      // Set up query sequence.
      ProteinSequence query = new ProteinSequence (queryfile);

      // Set up protein sequence database.
      ProteinDatabase database =
         new ProteinDatabase (databasefile, indexfile);
```

```
// Set up object to compute alignment statistics.
AlignmentStats stats =
   new DefaultAlignmentStats (database.getDatabaseLength());

// Set up list to hold alignments.
List<Alignment> alignments = new ArrayList<Alignment>();

// Set up object to perform alignments.
ProteinLocalAlignment aligner =
   new ProteinLocalAlignmentSeq();
aligner.setQuerySequence (query, 0);

long t2 = System.currentTimeMillis();

// Align query sequence against every subject sequence.
for (long id = 0; id < database.getProteinCount(); ++ id)
   {
   ProteinSequence subject =
      database.getProteinSequence (id);
   aligner.setSubjectSequence (subject, id);
   Alignment a = aligner.align();
   if (stats.eValue (a) <= expect)
      {
      alignments.add (a);
      }
   }

long t3 = System.currentTimeMillis();

// Sort alignments into descending order of score.
Collections.sort (alignments);

// Set up alignment printer.
AlignmentPrinter printer =
   new AlignmentPrinter (System.out, stats);

// Print query sequence.
System.out.println ("Query Description:");
System.out.println (query.description());
System.out.println ("Length = "+query.length());
System.out.println();

// Print summary of each alignment.
for (Alignment a : alignments)
   {
   printer.printSummary
```

```
            (a, database.getProteinSequence (a.getSubjectId()));
         }
      System.out.println();

      // Print details of each alignment.
      for (Alignment a : alignments)
         {
         printer.printDetails
            (a, query,
             database.getProteinSequence (a.getSubjectId()));
         }

      // Print various information about the alignment procedure.
      System.out.println ("Query file: "+queryfile);
      System.out.println ("Database file: "+databasefile);
      System.out.println ("Database index file: "+indexfile);
      System.out.println ("Number of sequences: "+
         database.getProteinCount());
      System.out.println ("Number of matches: "+alignments.size());
      System.out.println ("Query length: "+query.length());
      System.out.println ("Database length: "+
         database.getDatabaseLength());
      stats.print (System.out);
      System.out.println();

      // All done.
      database.close();

      long t4 = System.currentTimeMillis();
      System.out.println ((t2-t1)+" msec pre");
      System.out.println ((t3-t2)+" msec calc");
      System.out.println ((t4-t3)+" msec post");
      System.out.println ((t4-t1)+" msec total");
      }
   }
```

## 37.5 Parallel Program, Version 1

Now that we have a sequential version of the protein sequence database-querying program, we can turn our attention to a parallel version. Recall the two design strategies for a parallel data-set querying program: replicate the data set and partition the query, or partition the data set and replicate the query. The first design decision is which strategy to use. In this application the data set is large and the query is comparatively small, so the second strategy makes the most sense. Each parallel processor will align the query sequence with a subset of the database sequences.

In a hybrid parallel program, each process running on one backend node has multiple threads; each thread runs on one CPU of the node and can access the node's shared memory. The second design decision is what the threads of each process will do. We could go in either of two directions:

- All the threads cooperate to align the query sequence with one subject sequence in an SMP parallel fashion, each thread computing a subset of the alignment.
- Each thread does the complete alignment of the query sequence with one subject sequence. Different threads do alignments with different subject sequences in parallel.

We'll develop two parallel programs; each will use one of the preceding design approaches. It will prove instructive to compare the two programs' performance.

Figure 37.10 shows the overall design of the first parallel program. Each process works on one subject sequence at a time from the protein sequence database. Within each process, all the threads work in parallel to compute the alignment between the subject sequence and the query sequence. The process then goes to the next subject sequence. Once all the alignments have been computed, one process prints a list of the matching proteins (those with alignment scores below the *E*-value threshold).

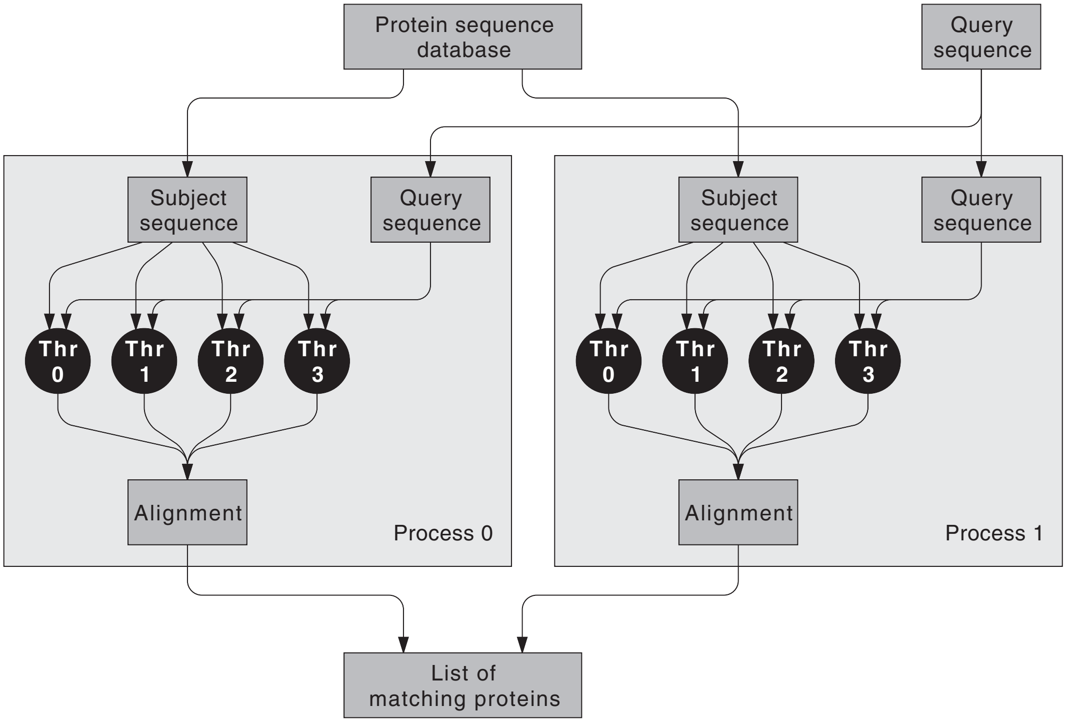


**Figure 37.10** First hybrid parallel program design with two processes and four threads per process

We need an SMP parallel version of the Smith-Waterman local alignment algorithm that we can use as a building block in the hybrid parallel program. To compute the correct answer, the SMP parallel version must obey the Smith-Waterman algorithm's sequential dependencies. Each element of the scoring matrix *S*[*i*, *j*] depends on the element above, *S*[*i*–1, *j*]; the element to the left, *S*[*i*, *j*–1]; and the element above and to the left, *S*[*i*–1, *j*–1] (Figure 37.11).

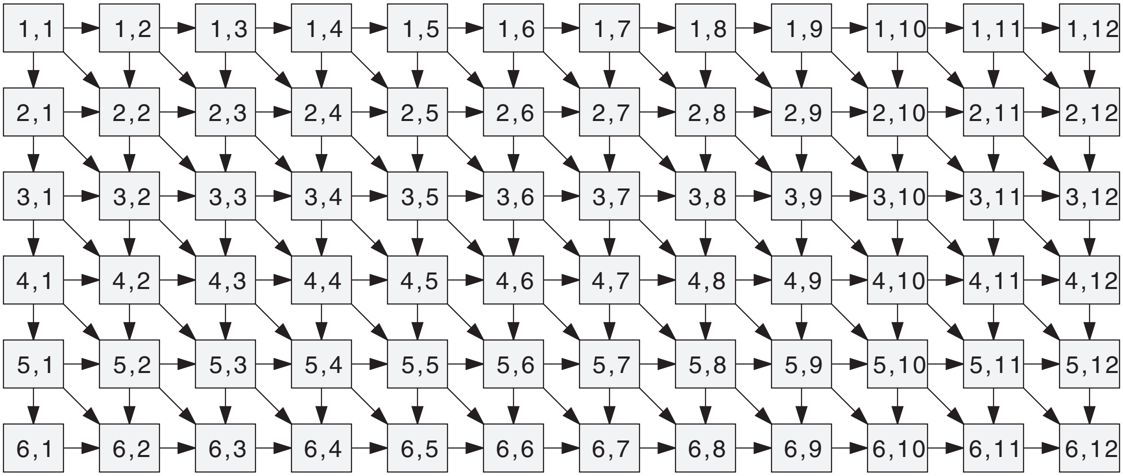


**Figure 37.11** Sequential dependencies in Smith-Waterman

Suppose we partition the columns of *S* among the parallel threads. Lumping each thread's columns in one row into a block results in the inter-block sequential dependencies shown in Figure 37.12.

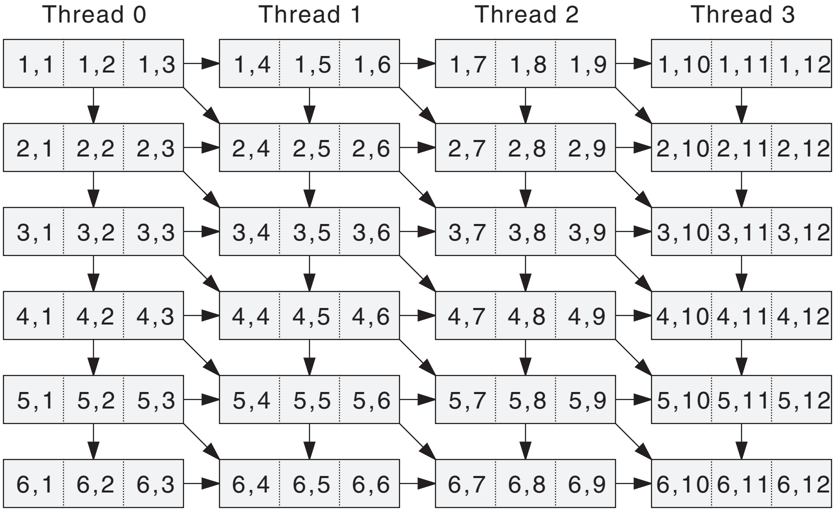
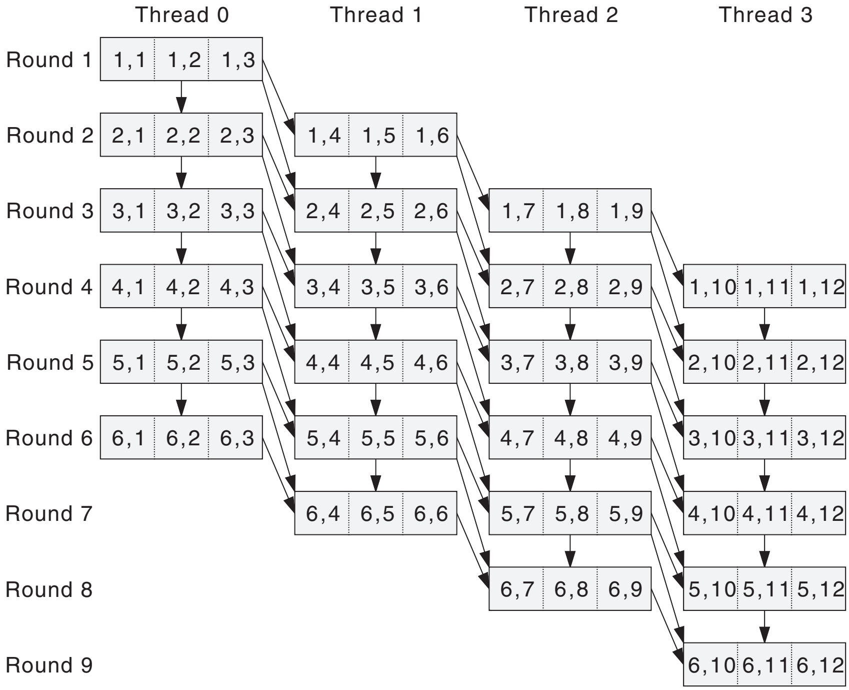


**Figure 37.12** Inter-block sequential dependencies in Smith-Waterman

Where is the parallelism? It still looks as though the columns in each row must be computed from left to right, so it doesn't seem possible that the threads can compute their blocks of columns simultaneously. However, suppose we shift the picture just a little (Figure 37.13).



| | Thread 0 | Thread 1 | Thread 2 | Thread 3 |
|---|---|---|---|---|
| Round 1 | 1,1 1,2 1,3 | | | |
| Round 2 | 2,1 2,2 2,3 | 1,4 1,5 1,6 | | |
| Round 3 | 3,1 3,2 3,3 | 2,4 2,5 2,6 | 1,7 1,8 1,9 | |
| Round 4 | 4,1 4,2 4,3 | 3,4 3,5 3,6 | 2,7 2,8 2,9 | 1,10 1,11 1,12 |
| Round 5 | 5,1 5,2 5,3 | 4,4 4,5 4,6 | 3,7 3,8 3,9 | 2,10 2,11 2,12 |
| Round 6 | 6,1 6,2 6,3 | 5,4 5,5 5,6 | 4,7 4,8 4,9 | 3,10 3,11 3,12 |
| Round 7 | | 6,4 6,5 6,6 | 5,7 5,8 5,9 | 4,10 4,11 4,12 |
| Round 8 | | | 6,7 6,8 6,9 | 5,10 5,11 5,12 |
| Round 9 | | | | 6,10 6,11 6,12 |

**Figure 37.13** Rounds of parallel computation in Smith-Waterman

There are *M*+*K*–1 rounds of computation, where *M* is the query sequence length (scoring matrix height) and *K* is the number of threads. In round *i*, thread *k* computes its own block of columns, from left to right, in row *i*–*k*; except if *i*–*k* < 1 or *i*–*k* > *M*, thread *k* does nothing in round *i*. With the computation arranged this way, there are inter-block sequential dependencies between rounds, but there are no inter-block sequential dependencies within a round. Therefore, the threads can compute their blocks in parallel in each round. To enforce the sequential dependency between rounds, the threads must do a barrier wait at the end of each round.

Here is the source code for class ProteinLocalAlignmentSmp, another subclass of abstract base class ProteinLocalAlignment. This subclass implements the `align()` method in the SMP parallel fashion described earlier. The parallel thread team is created in the main program, is supplied as a constructor argument, and is reused each time the `align()` method is called.

```
package edu.rit.compbio.seq;
import edu.rit.pj.ParallelRegion;
import edu.rit.pj.ParallelTeam;
import edu.rit.util.Range;
public class ProteinLocalAlignmentSmp
```

```
extends ProteinLocalAlignment
{
private ParallelTeam team;

/**
 * Construct a new protein sequence local alignment object.
 */
public ProteinLocalAlignmentSmp
   (ParallelTeam team)
   {
   super();
   if (team == null)
      {
      throw new NullPointerException
         ("ProteinLocalAlignmentSmp(): team is null");
      }
   this.team = team;
   }

/**
 * Align the query sequence and the subject sequence.
 */
public Alignment align()
   throws Exception
   {
   // Verify preconditions.
   if (A == null)
      {
      throw new IllegalStateException
         ("ProteinLocalAlignmentSmp.align(): Query sequence "+
          "not set");
      }
   if (B == null)
      {
      throw new IllegalStateException
         ("ProteinLocalAlignmentSmp.align(): Subject sequence "+
          "not set");
      }
   final int M = A.length - 1;
   final int N = B.length - 1;
   final int K = team.getThreadCount();
   final int lastRound = M + K - 1;
```

Each parallel team thread keeps track of the alignment finish point—that is, the row and column of the largest element—in the thread's own slice of the scoring matrix. After the threads have finished filling in the scoring matrix, the per-thread finish points must be reduced together. Here is the finish point shared reduction variable. It is an instance of class FinishPoint, a small helper class that is defined later.

```
// Initialize global finish point reduction variable.
final FinishPoint gblfp = new FinishPoint();
gblfp.maxScore = 0;
gblfp.theQueryFinish = 0;
gblfp.theSubjectFinish = 0;

// Do the Smith-Waterman algorithm in the parallel thread
// team.
team.execute (new ParallelRegion()
   {
   public void run() throws Exception
      {
      int threadIndex = getThreadIndex();

      // Determine range of columns for this thread.
      Range range =
         new Range (1, N) .subrange (K, threadIndex);
      int jlb = range.lb();
      int jub = range.ub();

      // Initialize per-thread finish point.
      int maxScore = 0;
      int theQueryFinish = 0;
      int theSubjectFinish = 0;

      // Do all rounds.
      for (int round = 1; round <= lastRound; ++ round)
         {
         // Row for this thread in this round is round number
         // offset by thread index. If row is out of bounds,
         // do nothing this round.
         int i = round - threadIndex;
         if (1 <= i && i <= M)
            {
            int A_i = A[i];
            int[] delta_A_i = delta[A_i];
            int[] S_im1 = S[i-1];
            int[] S_i = S[i];
            int[] GA_im1 = GA[i-1];
            int[] GA_i = GA[i];
            int[] GB_i = GB[i];
            int B_j, S_i_j, GA_i_j, GB_i_j;
            // Do only this thread's columns.
            for (int j = jlb; j <= jub; ++ j)
               {
               B_j = B[j];
               GA_i_j = S_im1[j] + g;
               GA_i_j = Math.max (GA_i_j, GA_im1[j] + h);
               GB_i_j = S_i[j-1] + g;
               GB_i_j = Math.max (GB_i_j, GB_i[j-1] + h);
               S_i_j = S_im1[j-1] + delta_A_i[B_j];
               S_i_j = Math.max (S_i_j, GA_i_j);
               S_i_j = Math.max (S_i_j, GB_i_j);
               S_i_j = Math.max (S_i_j, 0);
```

Here, the thread updates its own per-thread alignment finish point.

```
            if (S_i_j > maxScore)
               {
               maxScore = S_i_j;
               theQueryFinish = i;
               theSubjectFinish = j;
               }
            S_i[j] = S_i_j;
            GA_i[j] = GA_i_j;
            GB_i[j] = GB_i_j;
            }
         }

      // Wait for all threads to complete this round.
      barrier();
      }
```

Here, the thread reduces its per-thread alignment finish point into the shared reduction variable, by calling a method on the `gblfp` object.

```
   // After all rounds, reduce per-thread finish point
   // into global finish point.
   gblfp.setToBest
      (maxScore, theQueryFinish, theSubjectFinish);
   }
});
```

Because the traceback takes much less time than filling in the scoring matrix, the traceback is done outside the parallel region in a single thread.

```
   // Do the traceback in a single thread, starting from global
   // finish point.
   return computeTraceback
      (gblfp.maxScore,
       gblfp.theQueryFinish,
       gblfp.theSubjectFinish);
   }
```

Here is the helper class FinishPoint. The `setToBest()` method is a reduction method called by each thread. Because multiple threads are updating the shared finish point object, the threads must synchronize with each other. We accomplish this by making the `setToBest()` method a synchronized method, which ensures that only one thread at a time will execute the method. This method remembers the finish point with the highest score; if more than one finish point has the same highest score, then this method remembers the finish point in the smallest row; if more than one finish point has the same highest score in the same smallest row, then this method remembers the finish point in the smallest column. This behavior yields the same finish point as the sequential version.

```
private static class FinishPoint
   {
   // Alignment score.
   public int maxScore;

   // Query sequence index.
   public int theQueryFinish;

   // Subject sequence index.
   public int theSubjectFinish;

   // Set this finish point to the best of itself and the given
   // finish point. Multiple thread safe method.
   public synchronized void setToBest
      (int maxScore,
       int theQueryFinish,
       int theSubjectFinish)
      {
      if ((maxScore > this.maxScore) ||
         (maxScore == this.maxScore &&
            theQueryFinish < this.theQueryFinish) ||
         (maxScore == this.maxScore &&
            theQueryFinish == this.theQueryFinish &&
            theSubjectFinish < this.theSubjectFinish))
         {
         this.maxScore = maxScore;
         this.theQueryFinish = theQueryFinish;
```

```
            this.theSubjectFinish = theSubjectFinish;
            }
         }
      }
   }
```

Now that we have class ProteinLocalAlignmentSmp that can harness all the CPUs in one SMP node to do one alignment in parallel, we need a main program to partition the subject sequences from the protein database among the nodes of the hybrid parallel computer. Because the scoring matrix size, and therefore the time needed to fill it in, depends on the subject sequence length, and because the protein database has sequences of different lengths, different alignments take different amounts of time to compute, and load balancing is required. As usual, we will use the *master-worker pattern* for load balancing. The master in process 0 sends ranges of protein indexes to the workers. Following the *parallel input files pattern,* the worker reads each subject sequence from the protein database file, computes each alignment in an SMP parallel fashion, accumulates a list of Alignment objects having an *E*-value below the specified threshold, and sends the list back to the master. Because we designed class Alignment to be serializable, and because the class we are using for the list (java.util.ArrayList) is also serializable, the program has no trouble sending and receiving messages containing a list of alignments. Once all the workers have finished, process 0 sorts the alignments and prints the results.

Here is the source code for class FindProteinHyb, the first hybrid parallel version. Its command-line arguments are the same as the sequential version. Every backend node must be able to read the same query sequence file, protein database file, and database index file. The Java property `-Dpj.schedule` controls how the master apportions chunks of protein indexes among the workers.

```
package edu.rit.compbio.seq;
import edu.rit.mp.ObjectBuf;
import edu.rit.mp.buf.ObjectItemBuf;
import edu.rit.pj.Comm;
import edu.rit.pj.CommStatus;
import edu.rit.pj.LongSchedule;
import edu.rit.pj.ParallelRegion;
import edu.rit.pj.ParallelSection;
import edu.rit.pj.ParallelTeam;
import edu.rit.util.LongRange;
import java.io.File;
import java.io.IOException;
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
public class FindProteinHyb
   {
   // World communicator.
   static Comm world;
   static int size;
   static int rank;

   // Command line arguments.
   static File queryfile;
   static File databasefile;
   static File indexfile;
   static double expect;

   // Query sequence.
   static ProteinSequence query;

   // Protein sequence database.
   static ProteinDatabase database;

   // Object to compute alignment statistics.
   static AlignmentStats stats;

   // List of alignments found.
   static List<Alignment> alignmentsFound;

   /**
    * Main program.
    */
   public static void main
      (String[] args)
      throws Exception
      {
      // Start timing.
      long t1 = System.currentTimeMillis();

      // Initialize world communicator.
      Comm.init (args);
      world = Comm.world();
      size = world.size();
      rank = world.rank();

      // Parse command line arguments.
      if (3 > args.length || args.length > 4) usage();
      queryfile = new File (args[0]);
      databasefile = new File (args[1]);
      indexfile = new File (args[2]);
      expect = 10.0;
      if (args.length == 4) expect = Double.parseDouble (args[3]);
```

```
// Set up query sequence.
query = new ProteinSequence (queryfile);

// Set up protein sequence database.
database = new ProteinDatabase (databasefile, indexfile);

// Set up object to compute alignment statistics.
stats = new DefaultAlignmentStats
   (database.getDatabaseLength());

// Set up list of alignments found.
alignmentsFound = new ArrayList<Alignment>();

long t2 = System.currentTimeMillis();

// In process 0, run the master and the worker in separate
// threads.
if (rank == 0)
   {
   new ParallelTeam(2).execute (new ParallelRegion()
      {
      public void run() throws Exception
         {
         execute (new ParallelSection()
            {
            public void run() throws Exception
               {
               masterSection();
               }
            },
         new ParallelSection()
            {
            public void run() throws Exception
               {
               workerSection();
               }
            });
         }
      });
   }

// In processes 1 and up, run just the worker section.
else
   {
   workerSection();
   }
```

```
         long t3 = System.currentTimeMillis();

         // Process 0 does the postprocessing.
         if (rank == 0)
            {
            // Sort alignments into descending order of score.
            Collections.sort (alignmentsFound);

            // Set up alignment printer.
            AlignmentPrinter printer =
               new AlignmentPrinter (System.out, stats);

            // Print query sequence.
            System.out.println ("Query Description:");
            System.out.println (query.description());
            System.out.println ("Length = "+query.length());
            System.out.println();

            // Print summary of each alignment.
            for (Alignment a : alignmentsFound)
               {
               printer.printSummary
                  (a, database.getProteinSequence (a.getSubjectId()));
               }
            System.out.println();

            // Print details of each alignment.
            for (Alignment a : alignmentsFound)
               {
               printer.printDetails
                  (a, query,
                   database.getProteinSequence (a.getSubjectId()));
               }

            // Print various information about the alignment procedure.
            System.out.println
               ("Query file: "+queryfile);
            System.out.println
               ("Database file: "+databasefile);
            System.out.println
               ("Database index file: "+indexfile);
            System.out.println
               ("Number of sequences: "+database.getProteinCount());
            System.out.println
               ("Number of matches: "+alignmentsFound.size());
            System.out.println
               ("Query length: "+query.length());
            System.out.println
               ("Database length: "+database.getDatabaseLength());
            stats.print (System.out);
            System.out.println();

            long t4 = System.currentTimeMillis();
            System.out.println ((t2-t1)+" msec pre");
            System.out.println ((t3-t2)+" msec calc");
            System.out.println ((t4-t3)+" msec post");
            System.out.println ((t4-t1)+" msec total");
            }

         // Done using the protein sequence database.
         database.close();
         }

      /**
       * Perform the master section.
       */
      private static void masterSection()
         throws IOException
         {
         int worker;
         LongRange range;

         long t2 = System.currentTimeMillis();

         // Set up a schedule object.
         LongSchedule schedule = LongSchedule.runtime();
         schedule.start
            (size, new LongRange (0, database.getProteinCount()-1));

         // Send initial database index range to each worker. If range
         // is null, no more work for that worker. Keep count of
         // active workers.
         int activeWorkers = size;
         for (worker = 0; worker < size; ++ worker)
            {
            range = schedule.next (worker);
            world.send (worker, ObjectBuf.buffer (range));
            if (range == null) -- activeWorkers;
            }

         // Repeat until all workers have finished.
         while (activeWorkers > 0)
            {
            // Receive a message containing a list of zero or more
            // alignments from any worker.
            ObjectItemBuf<List<Alignment>> buf = ObjectBuf.buffer();
            CommStatus status = world.receive (null, buf);
            worker = status.fromRank;

            // Send next database index range to that specific worker.
            // If null, no more work.
            range = schedule.next (worker);
            world.send (worker, ObjectBuf.buffer (range));
            if (range == null) -- activeWorkers;

            // Add alignments to list.
            alignmentsFound.addAll (buf.item);
            }
         }

      /**
       * Perform the worker section.
       */
      private static void workerSection()
         throws Exception
         {
         // Set up object to perform alignments in multiple threads.
         ProteinLocalAlignment aligner =
            new ProteinLocalAlignmentSmp (new ParallelTeam());
         aligner.setQuerySequence (query, 0);

         // Process chunks from master.
         for (;;)
            {
            // Receive database index range from master. If null, no
            // more work.
            ObjectItemBuf<LongRange> rangeBuf = ObjectBuf.buffer();
            world.receive (0, rangeBuf);
            LongRange range = rangeBuf.item;
            if (range == null) break;
            long lb = range.lb();
            long ub = range.ub();

            // Set up list to hold alignments.
            List<Alignment> alignments = new ArrayList<Alignment>();

            // Align query sequence against every subject sequence.
            for (long id = lb; id <= ub; ++ id)
               {
               ProteinSequence subject =
                   database.getProteinSequence (id);
               aligner.setSubjectSequence (subject, id);
               Alignment a = aligner.align();
               if (stats.eValue (a) <= expect)
                  {
                  alignments.add (a);
                  }
               }

            // Send alignments back to master.
            world.send (0, ObjectBuf.buffer (alignments));
            }
         }
      }
```

We will put off examining the FindProteinHyb program's performance until we have developed the second parallel version and can compare the two.

## 37.6 Parallel Program, Version 2

For the second version of the hybrid parallel protein sequence database-querying program (Figure 37.14), we will retain the parallel data-set querying strategy: partition the data set, replicate the query. But this time, each thread of each process computes the complete alignment with one subject sequence. The thread then goes to the next subject sequence. Once all the alignments have been computed, one process prints a list of the matching proteins (those with alignment scores below the *E*-value threshold). To balance the load among the threads as well as the processes, we will use the *master-worker pattern with two-level scheduling.*

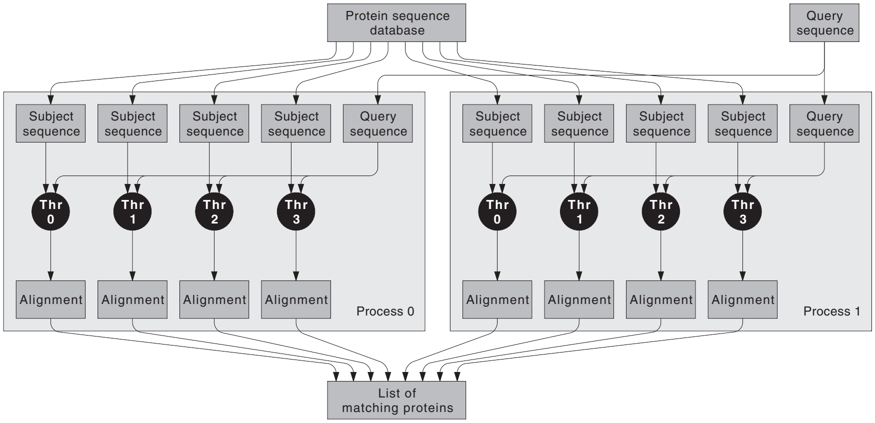


**Figure 37.14** Second hybrid parallel program design with two processes and four threads per process

The program's command-line arguments are the same as the sequential version, with one additional argument for the load-balancing schedule:

- Name of the file containing the query sequence in FASTA format.
- Name of the file containing the protein sequence database in FASTA format.
- Name of the database index file.
- Thread-level load-balancing schedule. If omitted, the default is a fixed schedule.
- *E*-value threshold for reporting a match. If omitted, the default is 10.

The Java property `-Dpj.schedule` gives the process-level load-balancing schedule.

Here is the source code for class FindProteinHyb2.

```
package edu.rit.compbio.seq;
import edu.rit.mp.ObjectBuf;
import edu.rit.mp.buf.ObjectItemBuf;
import edu.rit.pj.Comm;
import edu.rit.pj.CommStatus;
import edu.rit.pj.LongForLoop;
import edu.rit.pj.LongSchedule;
import edu.rit.pj.ParallelRegion;
import edu.rit.pj.ParallelSection;
import edu.rit.pj.ParallelTeam;
import edu.rit.util.LongRange;
import java.io.File;
import java.io.IOException;
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
public class FindProteinHyb2
   {
   // World communicator.
   static Comm world;
   static int size;
   static int rank;

   // Command line arguments.
   static File queryfile;
   static File databasefile;
   static File indexfile;
   static LongSchedule thrschedule;
   static double expect;

   // Query sequence.
   static ProteinSequence query;
```

```
// Protein sequence database.
static ProteinDatabase database;

// Object to compute alignment statistics.
static AlignmentStats stats;

// List of alignments found.
static List<Alignment> alignmentsFound;
```

The main program and `masterSection()` method are omitted because they are almost identical to the FindProteinHyb program. The only differences between the two programs are in parsing the thread-level schedule `(thrschedule)` from the fourth command-line argument, and in the `workerSection()` method.

```
private static void workerSection()
   throws Exception
   {
   // Set up parallel team.
   ParallelTeam team = new ParallelTeam();
   int K = team.getThreadCount();

   // Set up per-thread objects to perform alignments.
   final ProteinLocalAlignment[] aligner =
      new ProteinLocalAlignment [K];
   for (int i = 0; i < K; ++ i)
      {
      aligner[i] = new ProteinLocalAlignmentSeq();
      aligner[i].setQuerySequence (query, 0);
      }

   // Process chunks from master.
   for (;;)
      {
      // Receive database index range from master. If null, no
      // more work.
      ObjectItemBuf<LongRange> rangeBuf = ObjectBuf.buffer();
      world.receive (0, rangeBuf);
      LongRange range = rangeBuf.item;
      if (range == null) break;
      final long lb = range.lb();
      final long ub = range.ub();

      // Set up list to hold alignments.
      final List<Alignment> alignments =
         new ArrayList<Alignment>();
```

```
// Align query sequence against every subject sequence.
team.execute (new ParallelRegion()
   {
   public void run() throws Exception
      {
      execute (lb, ub, new LongForLoop()
         {
         // Per-thread variables plus padding.
         ProteinLocalAlignment thrAligner;
         List<Alignment> thrAlignments;
         long p0, p1, p2, p3, p4, p5, p6, p7;
         long p8, p9, pa, pb, pc, pd, pe, pf;

         // Initialize per-thread variables.
         public void start()
            {
            thrAligner = aligner[getThreadIndex()];
            thrAlignments = new ArrayList<Alignment>();
            }

         // Use thread-level loop schedule.
         public LongSchedule schedule()
            {
            return thrschedule;
            }

         // Do alignments.
         public void run (long first, long last)
            throws Exception
            {
            for (long id = first; id <= last; ++ id)
               {
               ProteinSequence subject =
                  database.getProteinSequence (id);
               thrAligner.setSubjectSequence
                  (subject, id);
               Alignment a = thrAligner.align();
               if (stats.eValue (a) <= expect)
                  {
                  thrAlignments.add (a);
                  }
               }
            }
```

```
                  // Reduce per-thread alignments into global
                  // alignments.
                  public void finish() throws Exception
                     {
                     region().critical (new ParallelSection()
                        {
                        public void run()
                           {
                           alignments.addAll (thrAlignments);
                           }
                        });
                     }
                  });
               }
            });

         // Send alignments back to master.
         world.send (0, ObjectBuf.buffer (alignments));
         }
      };
   }
```

## 37.7 Parallel Program Performance

Both hybrid parallel protein sequence database-querying programs were run on the “tardis” computer to compare their performance. The data set was the 385,721-sequence Swiss-Prot database, with a total of 138,434,015 amino acids. The query sequence was protein accession number P0C5C4, a 270-amino-acid protein from the tuberculosis bacterium. For both programs, a guided schedule was used to partition the database among the parallel processes. For the second program, the thread-level schedule was also a guided schedule.

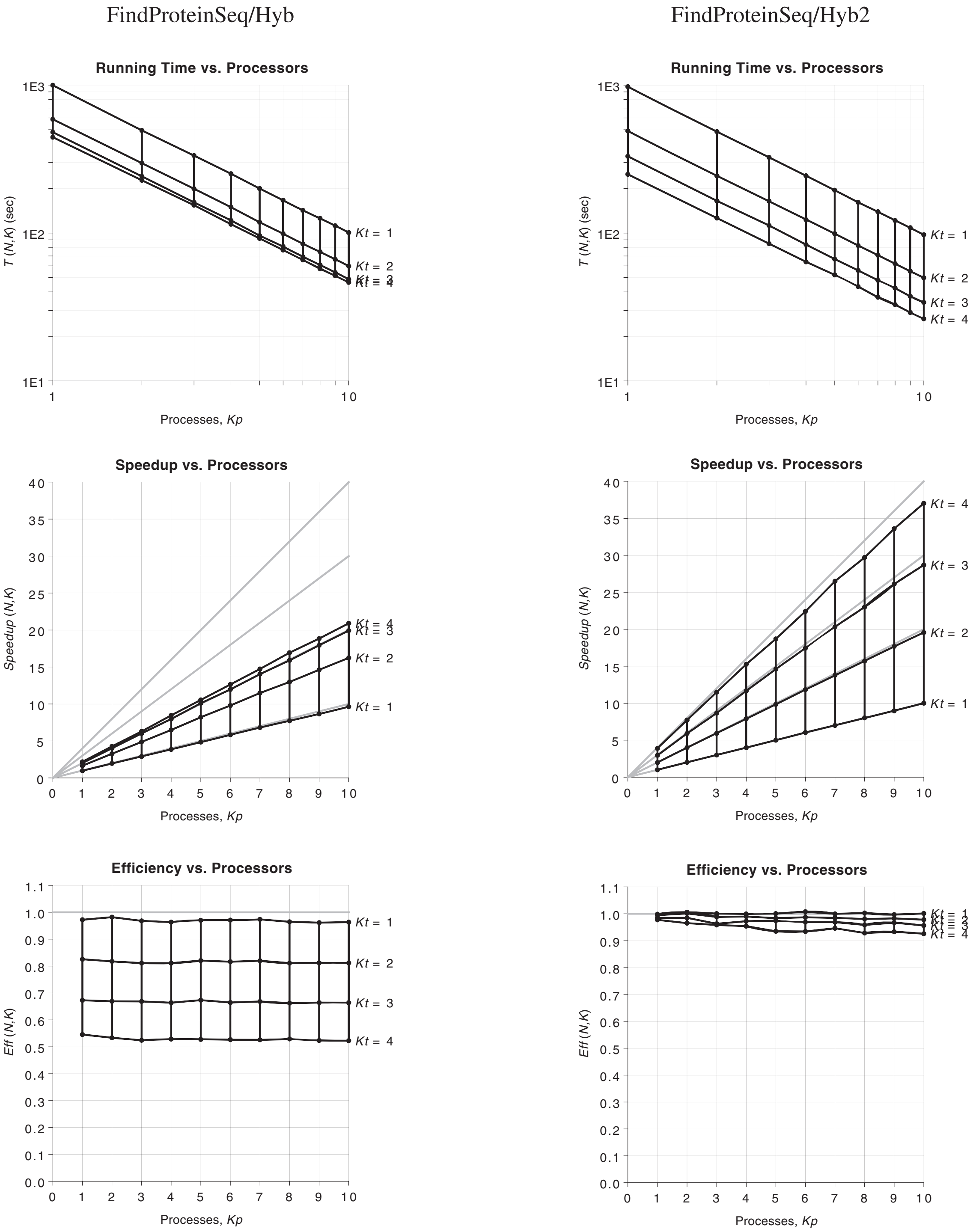


**Figure 37.15** Protein database-querying program running-time metrics

Table 37.2 (at the end of the chapter) lists, and Figure 37.15 plots, the two programs' running-time metrics for $Kp = 1$ to 10 processes and $Kt = 1$ to 4 threads per process. The metrics include the time to read all the subject sequences from the protein database file and to compute all the alignments, but do not include the time to sort the matching alignments and print the results (a matter of only a few seconds).

Both programs scale well as the number of processes increases. But only the second program scales well as the number of threads per process increases. The first program, where the threads in each process all work on the same alignment simultaneously, performs poorly with more than one thread per process—81 percent efficiency with two threads, 66 percent efficiency with three, 52 percent efficiency with four.

To gain some insight into the reason for this poor scalability, let's derive a rough model for the first program's running time in one process. We'll assume that all of the parallel version's extra time is due to thread synchronization, namely, the barrier waits in the SMP parallel `align()` method. The program does one barrier wait for every element of the query sequence (every row of the scoring matrix) and every subject sequence (every alignment). Therefore, the number of barrier waits was $270 \times 385{,}721 = 1.04 \times 10^8$. We'll further assume that the time to do a barrier wait is proportional to the number of threads in the parallel team $Kt$. Then the running time model is

$$T_{\text{par}} = T_{\text{seq}} / Kt + 1.04 \times 10^8 \cdot T_{\text{barrier}} \cdot Kt \qquad (37.6)$$

where $T_{\text{par}}$ is the parallel version's running time, $T_{\text{seq}}$ is the sequential version's running time, and $T_{\text{barrier}}$ is the time per thread to do a barrier wait. A bit of algebra tells us the following:

$$T_{\text{barrier}} = (T_{\text{par}} / Kt - T_{\text{seq}} / Kt^2) \div 1.04 \times 10^8 \qquad (37.7)$$

Plugging the first program's running-time data in one process into (37.7) gives these estimates for $T_{\text{barrier}}$:

| $T_{\text{seq}}$ (sec) | $Kt$ | $T_{\text{par}}$ (sec) | $T_{\text{barrier}}$ (sec) |
|---|---|---|---|
| $9.70\times10^2$ | 1 | $9.98\times10^2$ | $2.69\times10^{-7}$ |
| | 2 | $5.88\times10^2$ | $4.95\times10^{-7}$ |
| | 3 | $4.81\times10^2$ | $5.05\times10^{-7}$ |
| | 4 | $4.45\times10^2$ | $4.87\times10^{-7}$ |

The number of scoring matrix elements the program filled out was $270 \times 138{,}434{,}015 = 3.74 \times 10^{10}$, and the sequential version took $9.70 \times 10^2$ seconds to do it, so the time per matrix element was $9.70 \times 10^2$ sec $\div 3.74 \times 10^{10} = 2.59 \times 10^{-8}$ sec. Dropping the scientific notation, it takes 500 nanoseconds (nsec) per thread to do a barrier wait for $Kt \geq 2$, but only 25 nsec to calculate a scoring matrix element on the "tardis" computer. For a median-length 300-amino-acid subject sequence in the protein database, one scoring matrix row with one thread takes 7,500 nsec to calculate the matrix elements plus 269 nsec to do the barrier wait. But with four threads, one scoring matrix row takes 1,875 nsec to calculate the matrix elements plus 2,000 nsec to do the barrier wait. The thread synchronization takes longer than the computation! It's no wonder the efficiencies are so low.

When we studied cluster parallel programming, we said that to get good performance, there must be much more computation than communication. For SMP parallel programming, the analogous assertion is that there must be much more computation than *synchronization.* The subject sequence lengths in the Swiss-Prot database are just too short to get good SMP parallel performance; they don't result in enough computation relative to synchronization. On the other hand, the SMP parallel local alignment algorithm would scale better if the subject sequences were longer, say a median of 30,000 amino acids instead of 300.

## 37.8 Smith-Waterman vs. FASTA and BLAST

The FindProteinHyb program we have developed uses the Smith-Waterman local alignment algorithm. Published in 1981, the Smith-Waterman algorithm is guaranteed to find the optimal alignment between two sequences. However, its major drawback—especially when run on computers of 1980s vintage—is that it requires $O(n^2)$ time to fill out the complete scoring matrix, where $n$ is the length of the sequences being aligned, as well as $O(n^2)$ memory to store the scoring matrix.

Alternatives to Smith-Waterman, such as FASTA (first published in 1985) and BLAST (first published in 1990), addressed this drawback by using *heuristic* algorithms instead of an exact algorithm. Although not guaranteed to find optimal alignments, the FASTA and BLAST heuristics only require $O(n)$ time, and therefore run much faster than Smith-Waterman—especially on large protein sequence databases. While FASTA and BLAST have been refined and improved since their introduction, they are still essentially heuristic programs.

It's been nearly 30 years since Smith and Waterman published their algorithm. With parallel computers now becoming widespread and large main memories the norm, protein database-querying programs that find optimal alignments have become viable again and may start to see more use.

## 37.9 For Further Information

On biological sequence alignment algorithms in general:

- B. Jackson and S. Aluru. Pairwise sequence alignment. In S. Aluru, editor. *Handbook of Computational Molecular Biology.* Chapman & Hall/CRC, 2006, Chapter 1.

On the Smith-Waterman local alignment algorithm, in particular:

- T. Smith and M. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147:195–197, 1981.

On the Point Accepted Mutation (PAM) family of substitution matrices:

- M. Dayhoff, R. Schwartz, and B. Orcutt. A model of evolutionary change in proteins. In M. Dayhoff, editor. *Atlas of Protein Sequence and Structure*. National Biomedical Research Foundation, 1979, pages 345–352.

- M. Dayhoff and R. Schwartz. Matrices for detecting distant relationships. In M. Dayhoff, editor. *Atlas of Protein Sequence and Structure*. National Biomedical Research Foundation, 1979, pages 353–358.
- The PAM matrices: ftp://ftp.ncbi.nih.gov/blast/matrices/

On the Block Substitution Matrix (BLOSUM) family of substitution matrices:

- S. Henikoff and J. Henikoff. Amino acid substitution matrices from protein blocks. *Proceedings of the National Academy of Sciences of the USA*, 89(22):10915–10919, November 15, 1992.
- The BLOSUM matrices: ftp://ftp.ncbi.nih.gov/blast/matrices/

On the formulas for the *E*-value and bit score:

- S. Karlin and S. Altschul. Methods for assessing the statistical significance of molecular sequence features by using general scoring schemes. *Proceedings of the National Academy of Sciences of the USA*, 87(6):2264–2268, March 1987.
- S. Altschul and W. Gish. Local alignment statistics. *Methods in Enzymology*, 266:460–480, 1996.
- National Center for Biotechnology Information. The statistics of sequence similarity scores. http://www.ncbi.nlm.nih.gov/BLAST/tutorial/Altschul-1.html
- National Center for Biotechnology Information. The statistics of PSI-BLAST scores. http://www.ncbi.nlm.nih.gov/BLAST/tutorial/Altschul-3.html

On the Universal Protein Resource (UniProt):

- UniProt Home Page. http://www.uniprot.org/

On FASTA:

- D. Lipman and W. Pearson. Rapid and sensitive protein similarity searches. *Science*, 227(4693):1435–1441, March 22, 1985.
- W. Pearson and D. Lipman. Improved tools for biological sequence comparison. *Proceedings of the National Academy of Sciences of the USA*, 85(8):2444–2448, April 1988.
- FASTA Sequence Comparison at the U. of Virginia. http://fasta.bioch.virginia.edu/

On BLAST:

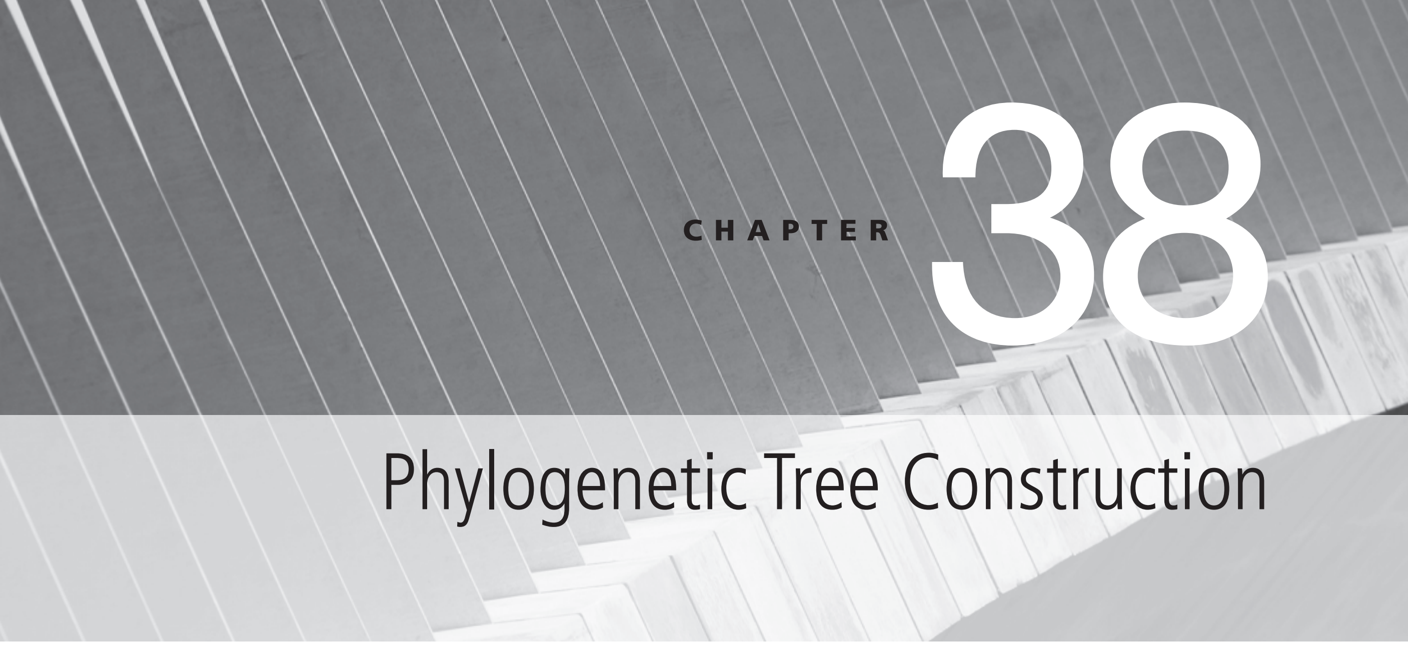
- S. Altschul, W. Gish, W. Miller, E. Myers, and D. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215(3):403–410, October 5, 1990.
- S. Altschul, T. Madden, A. Schäffer, J. Zhang, Z. Zhang, W. Miller, and D. Lipman. Gapped BLAST and PSI-BLAST: a new generation of protein database search programs. *Nucleic Acids Research*, 25(17):3389–3402, September 1, 1997.

- National Center for Biotechnology Information BLAST Home Page. http://blast.ncbi.nlm.nih.gov/

**Table 37.2** Protein database-querying program running-time metrics

| FindProteinSeq/FindProteinHyb | | | | | | FindProteinSeq/FindProteinHyb2 | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| *Kp* | *Kt* | *T* | *Spdup* | *Eff* | *EDSF* | *Kp* | *Kt* | *T* | *Spdup* | *Eff* | *EDSF* |
| seq | seq | 970139 | | | | seq | seq | 975370 | | | |
| 1 | 1 | 997795 | 0.972 | 0.972 | | 1 | 1 | 976808 | 0.999 | 0.999 | |
| 1 | 2 | 587690 | 1.651 | 0.825 | 0.178 | 1 | 2 | 490235 | 1.990 | 0.995 | 0.004 |
| 1 | 3 | 480622 | 2.019 | 0.673 | 0.223 | 1 | 3 | 330219 | 2.954 | 0.985 | 0.007 |
| 1 | 4 | 444599 | 2.182 | 0.546 | 0.261 | 1 | 4 | 249502 | 3.909 | 0.977 | 0.007 |
| 2 | 1 | 493996 | 1.964 | 0.982 | -0.010 | 2 | 1 | 484903 | 2.011 | 1.006 | -0.007 |
| 2 | 2 | 296681 | 3.270 | 0.817 | 0.063 | 2 | 2 | 243550 | 4.005 | 1.001 | -0.001 |
| 2 | 3 | 241678 | 4.014 | 0.669 | 0.091 | 2 | 3 | 165055 | 5.909 | 0.985 | 0.003 |
| 2 | 4 | 227324 | 4.268 | 0.533 | 0.118 | 2 | 4 | 126327 | 7.721 | 0.965 | 0.005 |
| 3 | 1 | 334046 | 2.904 | 0.968 | 0.002 | 3 | 1 | 324920 | 3.002 | 1.001 | -0.001 |
| 3 | 2 | 199334 | 4.867 | 0.811 | 0.040 | 3 | 2 | 164494 | 5.930 | 0.988 | 0.002 |
| 3 | 3 | 161244 | 6.017 | 0.669 | 0.057 | 3 | 3 | 112554 | 8.666 | 0.963 | 0.005 |
| 3 | 4 | 154256 | 6.289 | 0.524 | 0.078 | 3 | 4 | 84815 | 11.500 | 0.958 | 0.004 |
| 4 | 1 | 251721 | 3.854 | 0.964 | 0.003 | 4 | 1 | 243966 | 3.998 | 0.999 | 0.000 |
| 4 | 2 | 149544 | 6.487 | 0.811 | 0.028 | 4 | 2 | 123192 | 7.917 | 0.990 | 0.001 |
| 4 | 3 | 121774 | 7.967 | 0.664 | 0.042 | 4 | 3 | 83596 | 11.668 | 0.972 | 0.002 |
| 4 | 4 | 114707 | 8.458 | 0.529 | 0.056 | 4 | 4 | 63906 | 15.263 | 0.954 | 0.003 |
| 5 | 1 | 199959 | 4.852 | 0.970 | 0.001 | 5 | 1 | 194959 | 5.003 | 1.001 | -0.001 |
| 5 | 2 | 118233 | 8.205 | 0.821 | 0.021 | 5 | 2 | 99162 | 9.836 | 0.984 | 0.002 |
| 5 | 3 | 96057 | 10.100 | 0.673 | 0.032 | 5 | 3 | 66742 | 14.614 | 0.974 | 0.002 |
| 5 | 4 | 91985 | 10.547 | 0.527 | 0.044 | 5 | 4 | 52164 | 18.698 | 0.935 | 0.004 |
| 6 | 1 | 166516 | 5.826 | 0.971 | 0.000 | 6 | 1 | 161343 | 6.045 | 1.008 | -0.002 |
| 6 | 2 | 99059 | 9.794 | 0.816 | 0.017 | 6 | 2 | 82364 | 11.842 | 0.987 | 0.001 |
| 6 | 3 | 81099 | 11.962 | 0.665 | 0.027 | 6 | 3 | 55914 | 17.444 | 0.969 | 0.002 |
| 6 | 4 | 76788 | 12.634 | 0.526 | 0.037 | 6 | 4 | 43501 | 22.422 | 0.934 | 0.003 |
| 7 | 1 | 142332 | 6.816 | 0.974 | 0.000 | 7 | 1 | 139384 | 6.998 | 1.000 | 0.000 |
| 7 | 2 | 84518 | 11.478 | 0.820 | 0.014 | 7 | 2 | 70753 | 13.786 | 0.985 | 0.001 |
| 7 | 3 | 69122 | 14.035 | 0.668 | 0.023 | 7 | 3 | 47946 | 20.343 | 0.969 | 0.002 |
| 7 | 4 | 65856 | 14.731 | 0.526 | 0.031 | 7 | 4 | 36817 | 26.492 | 0.946 | 0.002 |
| 8 | 1 | 125712 | 7.717 | 0.965 | 0.001 | 8 | 1 | 121569 | 8.023 | 1.003 | -0.001 |
| 8 | 2 | 74759 | 12.977 | 0.811 | 0.013 | 8 | 2 | 62165 | 15.690 | 0.981 | 0.001 |
| 8 | 3 | 61029 | 15.896 | 0.662 | 0.020 | 8 | 3 | 42404 | 23.002 | 0.958 | 0.002 |
| 8 | 4 | 57301 | 16.931 | 0.529 | 0.027 | 8 | 4 | 32828 | 29.712 | 0.928 | 0.002 |
| 9 | 1 | 112107 | 8.654 | 0.962 | 0.001 | 9 | 1 | 108715 | 8.972 | 0.997 | 0.000 |
| 9 | 2 | 66362 | 14.619 | 0.812 | 0.012 | 9 | 2 | 55142 | 17.688 | 0.983 | 0.001 |
| 9 | 3 | 54093 | 17.935 | 0.664 | 0.018 | 9 | 3 | 37360 | 26.107 | 0.967 | 0.001 |
| 9 | 4 | 51488 | 18.842 | 0.523 | 0.025 | 9 | 4 | 29037 | 33.591 | 0.933 | 0.002 |
| 10 | 1 | 100693 | 9.635 | 0.963 | 0.001 | 10 | 1 | 97422 | 10.012 | 1.001 | 0.000 |
| 10 | 2 | 59768 | 16.232 | 0.812 | 0.010 | 10 | 2 | 49861 | 19.562 | 0.978 | 0.001 |
| 10 | 3 | 48707 | 19.918 | 0.664 | 0.016 | 10 | 3 | 33996 | 28.691 | 0.956 | 0.002 |
| 10 | 4 | 46407 | 20.905 | 0.523 | 0.022 | 10 | 4 | 26340 | 37.030 | 0.926 | 0.002 |

*This page intentionally left blank*



# CHAPTER 38

# Phylogenetic Tree Construction

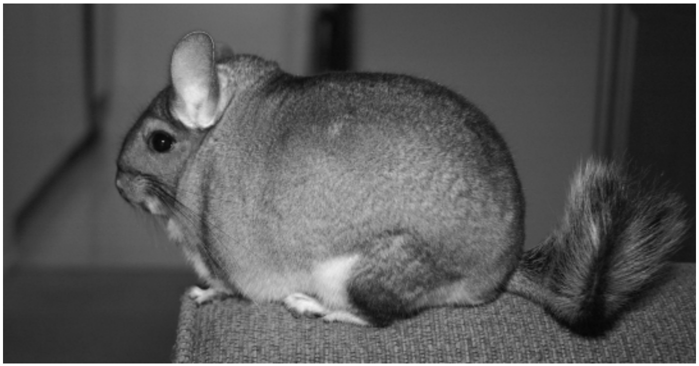
in which we consider how to determine organisms’ evolutionary relationships; we learn a technique for constructing a tree of relationships with the fewest number of mutations in the organism’s DNA; we design an SMP parallel program to compute the tree; and we observe how some problems can become too big—even for parallel computers

# CHAPTER 38 Phylogenetic Tree Construction

## 38.1 Phylogeny

In addition to studying the characteristics of protein molecules, as we saw in Chapter 37, biologists classify whole organisms and study the organisms' interrelationships. Biologists use a multilevel **taxonomy** to classify organisms, giving the category at each level in the taxonomy an abstruse Greek or Latin name. Here is the classification of the chinchilla (Figure 38.1), a small rodent native to the Andes Mountains of South America:

| *Level* | *Name* |
|---|---|
| Kingdom | Animalia |
| Phylum | Chordata |
| Class | Mammalia |
| Order | Rodentia |
| Family | Chinchillidae |
| Genus | Chinchilla |
| Species | lanigera |



Courtesy of Trurl66. http://commmons.wikimedia.org/wiki/Image:Chinchilla_lanigeral.jpg

**Figure 38.1** Chinchilla

The scientific name of this species is *Chinchilla lanigera,* abbreviated *C. lanigera,* which just means "woolly chinchilla" in Latin.

Consider four species: the chinchilla, the viscacha, the agouti, and the human. The viscacha is another South American rodent resembling a chinchilla; the agouti, also a rodent, lives in Central America and looks somewhat like a chinchilla; the human is not a rodent, lives all over the world, and only faintly resembles a chinchilla. Displaying these species' classifications all at once, we get a tree (Figure 38.2). Species with a closer physical resemblance to each other end up closer together in the tree.

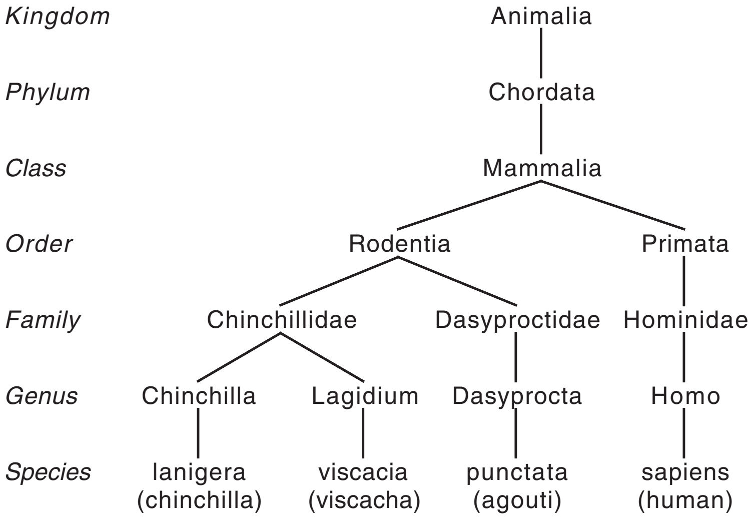


**Figure 38.2** Taxonomic tree of four mammalian species

Where taxonomy classifies species based on their physical characteristics, habitats, and other such external features, **phylogeny** classifies species based on their evolutionary ancestry. Because biologists have seldom been able to observe the evolutionary history directly, biologists usually *infer* the evolutionary history instead. Here's an oversimplified example: Because chinchillas and viscachas are physically very similar and live in nearby habitats in the Andes, we can infer that both rodents evolved from a common ancestor species sometime in the past. Because agoutis are somewhat different from chinchillas and viscachas physically, and they live in different habitats, we can infer that agoutis and the chinchilla/viscacha ancestor evolved from a more distant common ancestor. Although humans have some of the same characteristics as the other three (all four are hirsute, mammiferous, and viviparous, for example), but otherwise are rather dissimilar, we can infer that humans and the chinchilla/viscacha/agouti ancestor evolved from a common ancestor even further back. We can draw these inferred evolutionary relationships in a **phylogenetic tree** (Figure 38.3).

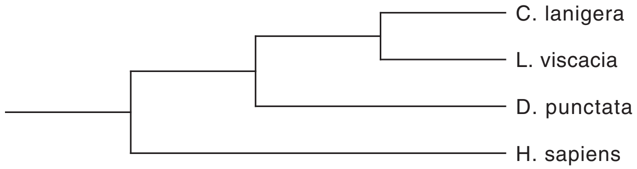


**Figure 38.3** Phylogenetic tree of four mammalian species

We'll draw the root of the tree at the left. As we move to the right, each interior node (where the tree splits into two branches) denotes a common ancestor. The tip nodes at the right are the extant species. The precise shape of the tree—the pattern of branches connecting the nodes—is called the tree's **topology**.

Biologists have long inferred phylogeny using the same data as for taxonomy—physical characteristics, habitat, and so on. However, evolution is driven by changes in an organism's genes. An altered gene may change the organism's physical characteristics, increasing or decreasing the organism's chances of survival. Organisms more likely to survive are more likely to pass on their altered genes to their offspring. Eventually, enough genetic changes accumulate to establish a new species. By studying the organisms' genes, the biologist can get a clearer picture of how the organisms evolved. Studying genes means, in turn, studying DNA.

**Deoxyribonucleic acid (DNA)** has been known since 1869, when Friedrich Miescher discovered the molecule in cell nuclei. Few suspected DNA's role as the carrier of genetic information until Oswald Avery, Colin MacLeod, and Maclyn McCarty showed in 1944 that chromosomes are made of DNA. In 1952, Martha Chase and Alfred Hershey proved that the DNA of the T2 bacteriophage virus carried the virus's genetic material. During this period, Rosalind Franklin and Maurice Wilkins used X-ray diffraction to amass data about DNA's structure. In 1953, Francis Crick and James Watson, basing their work in part on Franklin's and Wilkins's investigations, discovered the stereochemical structure of DNA—the famed *double helix* (Figure 38.4).

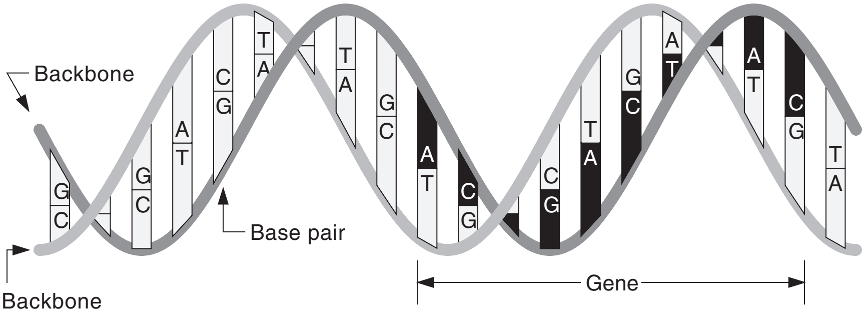


**Figure 38.4** Schematic representation of a DNA molecule

Each strand of a DNA molecule is composed of a long sequence of **nucleotides**, each nucleotide consisting of a **base** and a **sugar** (deoxyribose). A phosphorus atom binds every two adjacent nucleotides' sugars together, forming the DNA strand's helical sugar-phosphate **backbone**. The four bases found in DNA are **adenine**, **cytosine**, **guanine**, and **thymine**—A, C, G, and T. Each base on one strand binds to a base on the opposite strand, forming a **base pair**. A always binds to T, and C always binds to G, which explains Erwin Chargaff's 1950 discovery that DNA molecules contain equal amounts of adenine and thymine and equal amounts of cytosine and guanine.

A **gene** is a section of a DNA molecule that carries the information the cell needs to make one specific protein. The sequence of bases in the gene determines the sequence of amino acids in the protein; every three bases encode one amino acid. The mapping from base triplets to amino acids—the **genetic code**—was worked out by many researchers in the years following the discovery of the double helix.

Modern methods for deriving phylogenies use **DNA sequences**—strings of the letters A, C, G, and T. Like protein sequences, the DNA sequences of many genes are now known, including the complete **genomes**—all the genetic material—for some organisms. DNA sequence databases are readily available on the Web.

A phylogenetic tree construction method starts with a group of DNA sequences, with one sequence for each species at the tip nodes. For example, each sequence could be a different species' gene for a certain protein. The methods fall into three categories:

- A **likelihood method** finds a tree topology with the highest probability of occurring, given the sequences. We will not look at any likelihood methods.
- A **distance method** finds a tree topology, and assigns a length to each branch of the tree, such that the total length of the branches between any two sequences most closely matches the actual distance between the sequences. We will look at one simple distance method.
- A **parsimony method** finds a tree topology such that the number of genetic changes from the root of the tree to the tips is as small as possible, while still accounting for all the differences among the sequences. We will look at one parsimony method and use it as the basis for a parallel phylogenetic tree construction program.

## 38.2 Distances

Distance methods and parsimony methods both use distances. Before we can study these methods, we have to pin down the notion of distance.

The **distance** between two DNA sequences is a measure of the degree of difference between the sequences. Perhaps the simplest measure of distance was invented by Richard Hamming in 1950. The **Hamming distance** between two strings—in our case, between two DNA sequences—is the number of positions, or **sites**, at which the sequences differ. (Throughout this chapter, we'll assume that all the sequences are the same length.)

As an example, consider the DNA sequences for the insulin genes for our four mammalian species:

```
4 156
Chinchilla TTTGTCAACA AACATCTGTG CGGCTCACAC TTAGTGGATG CGCTATACCT
Viscacha   ATTGTCAACA AGCATCTGTG CGGCTCACAC TTAGTGGAGG CGCTATACAT
Agouti     TTTGTCAACC AGCATCTGTG CGGCTCCCAC TTAGTGGAGG CACTGTATAT
Human      TTTGTGAACC AACACCTGTG CGGCTCACAC CTGGTGGAAG CTCTCTACCT

GGTGTGTGGG GACAGAGGCT TCTTCTATAC ACCCATGGCC GGCATTGTGG ATCAGTGCTG
GGTGTGCAGG GATAAAGGCT TCTTCTATAC ACCCATGGAC GGCATTGTGG ATCAGTGCTG
GGCATGTGGG GACAAAGGCT TCTTCTATAC ACCGAAGGAC GGCATTGTGG ATCAGTGCTG
AGTGTGCGGG GAACGAGGCT TCTTCTACAC ACCCAAGACC GGCATTGTGG AACAATGCTG
```

```
TACCAGCATC TGCACACTCT ACCAGCTGGA GAACTACTGC AATTAG
TACCAGCATC TGCACACTTT ACCAGCTGGA GAACTACTGC AATTAG
TAACGGCATC TGCACATTCT ACCAGCTGCA GAGCTACTGC AACTAG
TACCAGCATC TGCTCCCTCT ACCAGCTGGA GAACTACTGC AACTAG
```

These sequences are in **PHYLIP format**. Joseph Felsenstein's Phylogeny Inference Package (PHYLIP) is a widely used set of programs for constructing phylogenetic trees and doing other kinds of analysis on genetic sequences. The first line gives the number of species and the sequence length. The next four lines give the name of each species (limited to 10 characters) and the initial section of each sequence. (Whitespace within the sequence is irrelevant.) Every remaining group of four lines gives the next section of each sequence.

The differences between the sequences become apparent when the sequences are represented this way:

```
4 156
Chinchilla TTTGTCAACA AACATCTGTG CGGCTCACAC TTAGTGGATG CGCTATACCT
Viscacha   A......... .G........ .......... ........G. ........A.
Agouti     .........C .G........ ......C... ........G. .A..G..TA.
Human      .....G...C ....C..... .......... C.G.....A. .T..C.....

GGTGTGTGGG GACAGAGGCT TCTTCTATAC ACCCATGGCC GGCATTGTGG ATCAGTGCTG
......CA.. ..T.A..... .......... ........A. .......... ..........
..CA...... ....A..... .......... ...G.A..A. .......... ..........
A.....C... ..AC...... .......C.. .....A.A.. .......... .A..A.....

TACCAGCATC TGCACACTCT ACCAGCTGGA GAACTACTGC AATTAG
.......... ........T. .......... .......... ......
..A.G..... ......T... ........C. ..G....... ..C...
.......... ...T.C.... .......... .......... ..C...
```

Here, a period stands for "the same character as the first sequence." The Hamming distances among the four example species' insulin sequences are listed in the following distance matrix:

| | Chinchilla | Viscacha | Agouti | Human |
|---|---|---|---|---|
| Chinchilla | 0 | 10 | 20 | 20 |
| Viscacha | 10 | 0 | 20 | 26 |
| Agouti | 20 | 20 | 0 | 31 |
| Human | 20 | 26 | 31 | 0 |

When inferring a phylogeny using a distance method, we want to assign a **branch length** to each branch of the tree. The branch length is supposed to represent the "amount of evolution" that has occurred between the ancestor species at one end of the branch and the descendant species at the other end. Specifically, the branch length should be the number of **state changes** that occurred along the branch. A state change is a genetic event that changed one base (A, C, G, or T) into a different base. However, the Hamming distance between two sequences is not necessarily the number of state changes that intervened between those two species during the course of evolution. Consider two species descending from a common ancestor. Along one branch, suppose one site experienced a state change from A to

C, and then later experienced another state change from C back to A. Along the other branch, suppose the same site experienced a state change from A to G. Then the Hamming distance ends up being 1, but the actual distance (number of state changes) was 3. In general, the Hamming distance between two sequences must be *corrected* to give the actual distance.

To do the corrections requires a model describing how states change. In 1969, Jukes and Cantor published a model in which state changes occur at random, but at a constant average rate at every site in every species, and each state changes to every other state with equal probability. Under the Jukes-Cantor model, the corrected distance (expected number of state changes) between two sequences is computed as

$$D_{JC} = -\frac{3}{4}\ L\ 1\text{n}\left(1 - \frac{4}{3} \cdot \frac{D_H}{L}\right) \tag{38.1}$$

where $D_{JC}$ is the Jukes-Cantor corrected distance, $D_H$ is the Hamming distance, and $L$ is the sequence length. For the four example sequences, with $L = 156$, the corrected distances are the following:

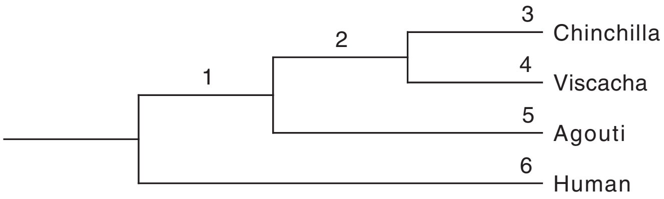
| | Chinchilla | Viscacha | Agouti | Human |
|---|---|---|---|---|
| Chinchilla | 0.00 | 10.45 | 21.93 | 21.93 |
| Viscacha | 10.45 | 0.00 | 21.93 | 29.40 |
| Agouti | 21.93 | 21.93 | 0.00 | 36.02 |
| Human | 21.93 | 29.40 | 36.02 | 0.00 |

Many, more complicated, models of state change have been published, but we will stick with the simple Jukes-Cantor model.

## 38.3 A Distance Method: UPGMA

Now that we can compute distances between sequences, let's use the distances to construct a phylogenetic tree using a distance method. The goal is to find a tree topology and branch lengths such that the total length of the branches between any two sequences most closely matches the actual distance between the sequences.

If we know the tree topology, then finding the branch lengths is easy. Here's how to do it. Suppose we use the following topology for the four example sequences. The tree branches are labeled 1 through 6.



Consider the first pair of sequences, chinchilla and viscacha. The Jukes-Cantor corrected distance between them is 10.45. Then, ideally, the following equation should hold,

$$x_3 + x_4 = 10.45 \tag{38.2}$$

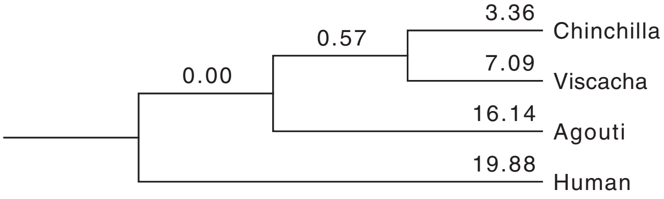
where $x_3$ is the length of branch 3 and $x_4$ is the length of branch 4. Repeating this for every pair of sequences gives the following system of equations:

$$\begin{aligned}
0 \cdot x_1 + 0 \cdot x_2 + 1 \cdot x_3 + 1 \cdot x_4 + 0 \cdot x_5 + 0 \cdot x_6 &= 10.45 \\
0 \cdot x_1 + 1 \cdot x_2 + 1 \cdot x_3 + 0 \cdot x_4 + 1 \cdot x_5 + 0 \cdot x_6 &= 21.93 \\
1 \cdot x_1 + 1 \cdot x_2 + 1 \cdot x_3 + 0 \cdot x_4 + 0 \cdot x_5 + 1 \cdot x_6 &= 21.93 \\
0 \cdot x_1 + 1 \cdot x_2 + 0 \cdot x_3 + 1 \cdot x_4 + 1 \cdot x_5 + 0 \cdot x_6 &= 21.93 \\
1 \cdot x_1 + 1 \cdot x_2 + 0 \cdot x_3 + 1 \cdot x_4 + 0 \cdot x_5 + 1 \cdot x_6 &= 29.40 \\
1 \cdot x_1 + 0 \cdot x_2 + 0 \cdot x_3 + 0 \cdot x_4 + 1 \cdot x_5 + 1 \cdot x_6 &= 36.02
\end{aligned} \tag{38.3}$$

Expressing Equation 38.3 in matrix notation gives

$$\mathbf{A} \cdot \mathbf{x} = \mathbf{d} \tag{38.4}$$

where **A** is a matrix of 0s and 1s indicating which branches lie on the path between each pair of sequences, **d** is a vector of the inter-sequence distances, and **x** is a vector of the branch lengths. Now all we have to do is find the vector **x** that minimizes the sum of the squared differences between the left and right sides of (38.4)—a *least-squares* solution. Furthermore, because it makes no sense for a branch length to be negative, we want to find a *nonnegative least-squares* solution. As we saw in Chapter 36, the Parallel Java Library provides class edu.rit.numeric.NonNegativeLeastSquares to do just that. The least-squares branch lengths for the example tree topology are shown in the following example; the squared error in the solution is 13.95:

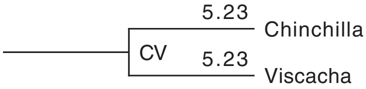


The preceding procedure finds the least-squares branch lengths, given the tree topology. But the tree we really want is the one that, of *all possible* tree topologies, gives the smallest squared error. We could try every topology and report the best, but, as we will see when we study parsimony methods, this can take a prohibitively long time.

As an alternative, we can use a **heuristic** algorithm that finds a solution without examining every topology. On the one hand, a heuristic algorithm is not guaranteed to find the tree with the smallest squared error. On the other hand, a heuristic algorithm typically takes much less time than finding the exact solution. As an example of a heuristic algorithm, we'll look at the Unweighted Pair Group Method with Arithmetic mean (UPGMA), published by Sokal and Michener in 1958.

UPGMA works by clustering the species into groups based on their distances. The algorithm keeps track of the number of species in each group. Initially, each species is by itself in a separate group. Here's how UPGMA constructs a tree for the four example species:

1. Find the two groups with the smallest distance—chinchilla and viscacha.
2. Join those groups into a new group; let's call it "CV." The new group has $n_i + n_j$ members, where $n_i$ and $n_j$ are the sizes of the groups that were joined. In this case $n_i + n_j = 2$.
3. Construct a three-node tree of chinchilla, viscacha, and their common ancestor CV. Each branch's length is half the distance between chinchilla and viscacha.



4. Compute the distance between CV and the remaining groups, agouti and human. The formula is

$$D_{(ij),k} = \left(\frac{n_i}{n_i + n_j}\right) D_{i,k} + \left(\frac{n_j}{n_i + n_j}\right) D_{j,k} \tag{38.5}$$

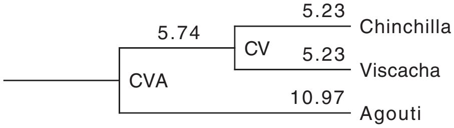
where $D$ is the distance matrix, $i$ and $j$ are the indexes of the two groups that were joined, $(ij)$ refers to the new group, $k$ is the index of one of the remaining groups, and $n_i$ and $n_j$ are the sizes of the original groups. In this case, the distance between CV and agouti is one-half the distance between chinchilla and agouti plus one-half the distance between viscacha and agouti, or 21.93. The distance between CV and human is 25.67.

5. Replace the two original groups in the distance matrix with the new group:

| | CV | Agouti | Human |
|---|---|---|---|
| CV | 0.00 | 21.93 | 25.67 |
| Agouti | 21.93 | 0.00 | 36.02 |
| Human | 25.67 | 36.02 | 0.00 |

6. Repeat Steps 1–5. This time, CV and agouti are the closest, resulting in a new common ancestor CVA. The branch length between CVA and agouti is one-half the distance between CV and agouti, namely 10.97. The branch length between CVA and *the species contained in CV* is also 10.97. But because

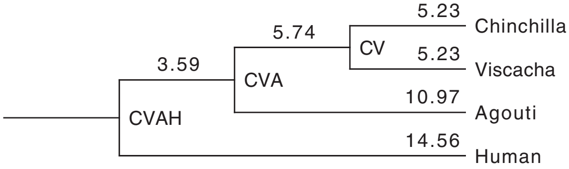
CV is already a branch length of 5.23 from chinchilla and viscacha, CVA is a branch length of 10.97 – 5.23 = 5.74 from CV.



The distance between CVA and human is two-thirds the distance between CV and human plus one-third the distance between agouti and human, or 29.12.

| | CVA | Human |
|---|---|---|
| CVA | 0.00 | 29.12 |
| Human | 29.12 | 0.00 |

7. Repeat Steps 1–5. Afterward, there's only one group left, so we're done. The final tree is the following:



Note that UPGMA produces branch lengths such that for each interior node, the total distance from the interior node to any tip node along one branch is the same as the total distance from the interior node to any tip node along the other branch. A tree with this property is called **ultrametric**.

How closely did the UPGMA tree reproduce the distances between the species? Here are the tree distances if you follow the branches:

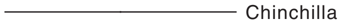
| | Chinchilla | Viscacha | Agouti | Human |
|---|---|---|---|---|
| Chinchilla | 0.00 | 10.46 | 21.94 | 29.12 |
| Viscacha | 10.46 | 0.00 | 21.94 | 29.12 |
| Agouti | 21.94 | 21.94 | 0.00 | 29.12 |
| Human | 29.12 | 29.12 | 29.12 | 0.00 |

UPGMA didn't manage to reproduce the original distances exactly. Also, the squared error in the branch lengths UPGMA computed is 99.28, which is larger than the least-squares branch lengths for this topology. Despite these shortcomings, UPGMA finds both a topology and branch lengths very quickly, and we'll use UPGMA as part of the parsimony methods we'll study next.

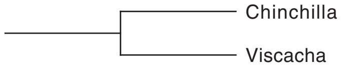
## 38.4 Maximum Parsimony Method

The maximum parsimony method for constructing a phylogenetic tree assumes that the tree that requires the fewest state changes to account for all the differences among the sequences—the *most parsimonious* tree—is the tree that best represents the species' evolutionary ancestry. In concept, the maximum parsimony method generates every possible tree topology, determines the minimum number of state changes—the **parsimony score**—for each tree, and reports the tree or trees with the smallest score.

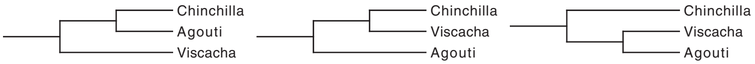
Generating all possible tree topologies is done by building up the trees one species at a time and adding each species at every possible position. Here's how it goes with our four example species. There's only one possible tree with one species:



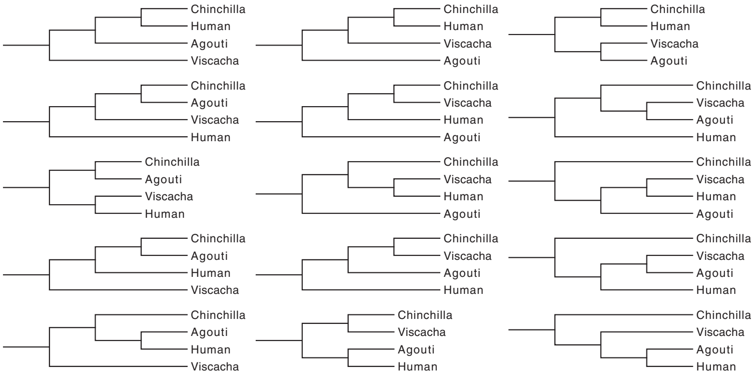
There's only one possible place in the preceding tree to add viscacha, as the sibling of chinchilla:



Now there are three places to add agouti—as the sibling of chinchilla, as the sibling of the chinchilla-viscacha ancestor, and as the sibling of viscacha—giving rise to three possible tree topologies of three species:



In each of the above preceding trees, there are five places to add human. We end up with 15 possible tree topologies of four species:



A pattern is starting to emerge. With 3 species, there are 3 possible topologies. With 4 species, there are 3·5 topologies. With 5 species, there are 3·5·7 topologies. With 6 species, there are 3·5·7·9 topologies. In general, with $N$ species, there are $(2N-3)!!$ topologies. The "!!" stands for "double factorial," which is defined as follows:

$$\begin{aligned} x!! &= x(x-2)\ (x-4)\ \cdots\ 5\cdot 3\cdot 1,\ \text{if } x \text{ is odd} \\ x!! &= x(x-2)\ (x-4)\ \cdots\ 6\cdot 4\cdot 2,\ \text{if } x \text{ is even} \end{aligned} \tag{38.6}$$

The double factorial is like the regular factorial, except successive terms differ by 2 instead of by 1. (Note that "!!" does *not* mean "factorial of the factorial.")

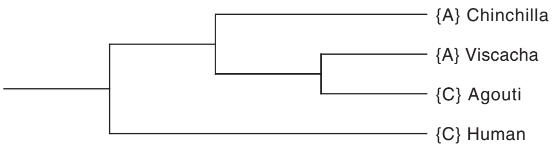
The double-factorial function becomes enormous very quickly. Here are the numbers of possible tree topologies for 1 to 20 species:

| $N$ | $(2N-3)!!$ | $N$ | $(2N-3)!!$ |
|---|---|---|---|
| 1 | $1.00 \times 10^0$ | 11 | $6.55 \times 10^8$ |
| 2 | $1.00 \times 10^0$ | 12 | $1.37 \times 10^{10}$ |
| 3 | $3.00 \times 10^0$ | 13 | $3.16 \times 10^{11}$ |
| 4 | $1.50 \times 10^1$ | 14 | $7.91 \times 10^{12}$ |
| 5 | $1.05 \times 10^2$ | 15 | $2.13 \times 10^{14}$ |
| 6 | $9.45 \times 10^2$ | 16 | $6.19 \times 10^{15}$ |
| 7 | $1.04 \times 10^4$ | 17 | $1.92 \times 10^{17}$ |
| 8 | $1.35 \times 10^5$ | 18 | $6.33 \times 10^{18}$ |
| 9 | $2.03 \times 10^6$ | 19 | $2.22 \times 10^{20}$ |
| 10 | $3.45 \times 10^7$ | 20 | $8.20 \times 10^{21}$ |

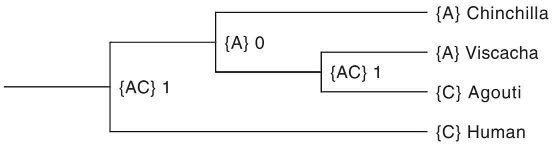
Clearly, finding the maximum parsimony tree for more than a dozen or so species is going to require a *lot* of computation just to go through all the possible topologies.

For a given tree topology, we need a way to determine the parsimony score—the minimum number of state changes to account for the differences among the sequences. Because of the enormous number of topologies we have to examine, it is crucial to determine the parsimony score as quickly as possible.

In 1971, Walter Fitch published an algorithm to compute a tree's parsimony score. The algorithm works its way from the tip nodes up to the root, associating a set of states with each site. At the tip nodes, the set of states contains just one element. For example, here is one topology with the four example species, looking at just the tenth site in each sequence:

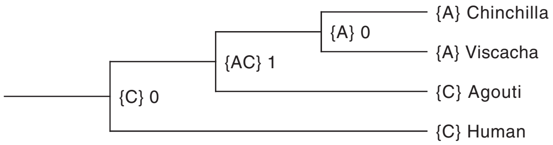


For each interior node, the state set is the intersection of the two child nodes' state sets; but if the intersection is empty, then the state set is the union of the two child nodes' state sets, and the parsimony score is increased by 1. Here's what we get for the example:



The total parsimony score is 2, meaning 2 state changes are required when going from the root to the tips. The Fitch algorithm doesn't tell us what the actual states are in each common ancestor (interior node), it just tells us how many state changes there are. Typically, multiple evolutionary histories compatible with the parsimony score are possible. One possible history assigns a state of A to each ancestor, and then agouti and human each require one state change. Another possible history assigns a state of C to each ancestor, and then chinchilla and viscacha each require one state change. To find the maximum parsimony tree, however, we only need to know the *number* of state changes, not their *locations* in the tree.

For the same species, different topologies may yield different parsimony scores. Here is another topology for the four example species, also looking at just the tenth site. This time the score is 1. A possible evolutionary history with one state change is C, C, A at the interior nodes.



The preceding example was for just one site. To compute the tree's complete parsimony score, we run the Fitch algorithm for each individual site and add up the results. The computation time is proportional to the number of tree nodes ($2N-1$, where $N$ is the number of species) times the number of sites ($L$, the sequence length).

As pointed out by Andrey Zharkikh, we can reduce that time by being smarter about which sites we include in the parsimony score computation. Consider our four example species. Most of the sites have the same state in all four sequences. Because these sites experienced no state changes, they contribute zero to the parsimony score, and eliminating them from consideration saves computation time. Here are the four example species with the unvarying sites omitted:

```
Chinchilla TCAATATATGACCGTGTGCAGTCTGCTGCAAACCAT
Viscacha   ACAGTATAGGACAGTGCATAATCTGATGCAAACTAT
Agouti     TCCGTCTAGAGTAGCATGCAATGAGATGAGAATCGC
Human      TGCACACGATCCCATGCGACGCCAACAACATCCCAC
```

We only have to look at 36 sites, not all 156—a 77 percent savings.

We can be smarter still. Consider the first site, with states of T, A, T, and T. Somewhere in the tree, this site must switch from T to A; everywhere else, this site stays at T. Therefore, this site will contribute 1 to the parsimony score *no matter what the tree topology is.* Also, consider the tenth site, with states of G, G, A, and T. Somewhere this site must switch from G to A; somewhere else, from G to T. Therefore, this site will contribute 2 to the score in every tree topology. To yield different scores in different topologies, a site must have at least two different states, and each state must appear in at least two different sequences—such as the site with states A, A, C, and C that we saw earlier. Such sites are called **informative sites**; the other sites are called **uninformative sites**. We can eliminate the uninformative sites from consideration, as long as we remember the number of state changes the uninformative sites contribute to the score (each uninformative site contributes *S*–1 state changes, where *S* is the number of unique states at the site). Here are the uninformative sites in the four example species, along with the amounts they contribute:

```
Chinchilla TCAATATATGACCGTGTGCAGTCTGCTGCAAACCAT
Viscacha   ACAGTATAGGACAGTGCATAATCTGATGCAAACTAT
Agouti     TCCGTCTAGAGTAGCATGCAATGAGATGAGAATCGC
Human      TGCACACGATCCCATGCGACGCCAACAACATCCCAC
           11  11112221 111 121 11 1 111111111 ← Uninformative sites
```

The uninformative sites contribute a total of 32 to every possible tree's parsimony score. When scoring trees, then, we only have to look at the 8 remaining informative sites, not all 156—a 95 percent savings in the computation time.

## 38.5 Maximum Parsimony with Exhaustive Search

We can now design a sequential program to find the maximum parsimony phylogenetic tree or trees for a series of DNA sequences using **exhaustive search**. The search is called "exhaustive" because it considers all (2*N* – 3)!! possible tree topologies. (It is also called exhaustive because when *N* gets larger than a dozen or so species, our patience gets exhausted waiting for the program to finish!) Figure 38.5 shows the classes from which the sequential phylogeny program is built and their "uses" relationships. (A→B means class A uses class B.) Unless otherwise stated, these classes are in package edu.rit.compbio.phyl. The main program is class PhylogenyParsExhSeq. We'll describe what every class does, but study the code for only a few key classes.

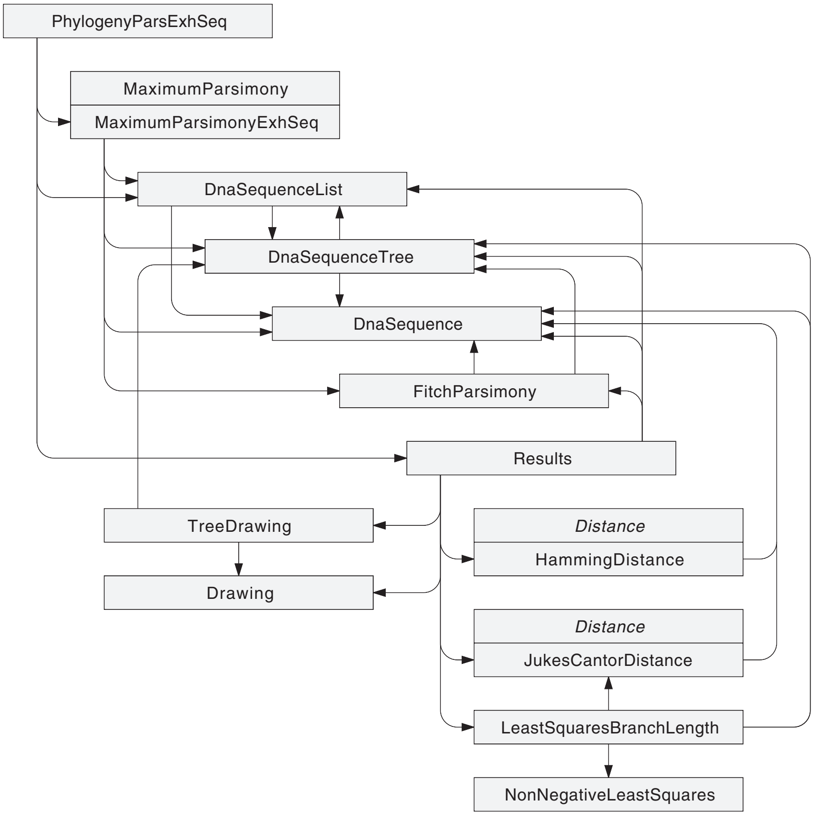


**Figure 38.5** PhylogenyParsExhSeq class relationships

- Class DnaSequence provides a sequence of sites, each site being a set of the states A, C, G, and T. This is used to represent the original DNA sequences at the tip nodes, as well as the state sets at the interior nodes in the Fitch parsimony scoring algorithm.
- Class DnaSequenceList provides a list of DnaSequences. A method is provided to read a list of DNA sequences from a file in PHYLIP format.
- Class DnaSequenceTree provides a phylogenetic tree of DnaSequences. The class is designed for building a tree by adding DNA sequences (tip nodes) one at a time. The `add()` method adds a new interior node with two child nodes. One child node is the node at a given position in the tree. The other child node is a new tip node with a given DNA sequence. Thus, the new tip node becomes the sibling of the existing node.
- Class FitchParsimony has a `computeScore()` method for computing a tree's parsimony score. It also has an `updateScore()` method for updating a tree's

parsimony score after a new DNA sequence has been added. The parsimony score can be updated by recalculating just the interior nodes along the path from the new tip node to the root; this takes less time than calculating all the nodes.

- Class MaximumParsimony is the abstract base class for an object that finds the most parsimonious tree or trees of the sequences in a DnaSequenceList. The base class has common fields and methods, as well as an abstract `findTrees()` method that can implemented in different ways. Here is the source code.

```
package edu.rit.compbio.phyl;
import java.util.List;
public abstract class MaximumParsimony
   {
   /**
    * List of DNA sequences. Input to the last findTrees() method
    * call.
    */
   public DnaSequenceList seqList;

   /**
    * Limit on the number of phylogenetic trees to store. Input to
    * the last findTrees() method call. The findTrees() method will
    * store at most this many maximum parsimony phylogenetic trees.
    */
   public int treeStoreLimit;

   /**
    * Initial bound for branch-and-bound search. Input to the last
    * findTrees() method call. The findTrees() method will only find
    * trees whose parsimony scores are less than or equal to the
    * bound.
    */
   public int initialBound;
```

We will look at branch-and-bound search after we've looked at exhaustive search.

```
   /**
    * List of maximum parsimony phylogenetic trees. Output from the
    * last findTrees() method call. Contains one or more tree
    * signatures representing the phylogenetic trees, all of which
    * have the same maximum parsimony score.
    */
   public List<int[]> treeList;
```

A **tree signature** is a compact way to designate a particular tree topology. The signature is an array of *N* positions, where *N* is the number of species. The signature at index *i* gives the position in the tree at which the *i*-th species is to be added. By adding the species, in order, to an empty tree at the positions given in the signature, the tree can be reconstructed. Later, we will use a tree signature to keep track of our place when generating all possible tree topologies.

```
/**
 * Maximum parsimony score. Output from the last findTrees()
 * method call. Contains the Fitch parsimony score of the
 * phylogenetic trees in treelist.
 */
public int score;

/**
 * Construct a new maximum parsimony phylogenetic tree
 * construction algorithm object.
 */
public MaximumParsimony()
   {
   }

/**
 * Find the maximum parsimony phylogenetic tree(s) for the given
 * DNA sequence list. The results are stored in the fields of
 * this object. The findTrees() method will store at most
 * treeStoreLimit maximum parsimony phylogenetic trees. The
 * findTrees() method will only find trees whose parsimony scores
 * are less than or equal to the initialBound.
 *
 * @param  seqList         DNA sequence list.
 * @param  treeStoreLimit  Maximum number of trees to store.
 * @param  initialBound    Initial bound for branch-and-bound
 *                         search.
 */
public abstract void findTrees
   (DnaSequenceList seqList,
    int treeStoreLimit,
    int initialBound)
   throws Exception;

/**
 * Returns the number of phylogenetic trees found by the last
 * findTrees() method call.
 *
 * @return  Number of trees.
```

```
    */
   public int length()
      {
      return treeList.size();
      }

   /**
    * Returns the phylogenetic tree found by the last findTrees()
    * method call corresponding to the given index. The returned
    * tree is newly created based on the tree signature.
    *
    * @param  i  Index in the range 0 .. length()-1.
    *
    * @return  Phylogenetic tree.
    */
   public DnaSequenceTree tree
      (int i)
      {
      return seqList.toTree (treeList.get (i));
      }
   }
```

- Class MaximumParsimonyExhSeq extends base class MaximumParsimony with the algorithm for an exhaustive search executed sequentially in a single thread. Here is the source code.

```
package edu.rit.compbio.phyl;
import java.util.ArrayList;
public class MaximumParsimonyExhSeq
   extends MaximumParsimony
   {
   /**
    * Construct a new maximum parsimony phylogenetic tree
    * construction algorithm object.
    */
   public MaximumParsimonyExhSeq()
      {
      }

   /**
    * Find the maximum parsimony phylogenetic tree(s) for the given
    * DNA sequence list. The results are stored in the fields of
    * this object. The findTrees() method will store at most
    * treeStoreLimit maximum parsimony phylogenetic trees. For
    * exhaustive search, the initialBound argument is ignored.
    *
    * @param  seqList         DNA sequence list.
    * @param  treeStoreLimit  Maximum number of trees to store.
    * @param  initialBound    Initial bound for branch-and-bound
    *                         search.
    */
   public void findTrees
      (DnaSequenceList seqList,
       int treeStoreLimit,
       int initialBound)
      {
      // Initialize.
      this.seqList = seqList;
      this.treeStoreLimit = treeStoreLimit;
      this.initialBound = initialBound;
      this.treeList = new ArrayList<int[]> (treeStoreLimit);
      this.score = Integer.MAX_VALUE;
      int L = seqList.seq(0).length();
      int N = seqList.length();
      int C = 2*N + 1;
```

To generate all possible tree topologies, we will add the DNA sequences one at a time. After adding each sequence at a certain position, we will save the resulting tree in the `treeStack` variable. `treeStack[0]` holds the tree after adding the first sequence to an empty tree, `treeStack[1]` holds the tree after adding the second sequence to `treeStack[0]`, `treeStack[2]` holds the tree after adding the third sequence to `treeStack[1]`, and so on.

```
      // Set up stack of DNA sequence trees.
      DnaSequenceTree[] treeStack = new DnaSequenceTree [N];
      for (int i = 0; i < N; ++ i)
         {
         treeStack[i] = new DnaSequenceTree (C);
         }
```

Each tree in `treeStack` is paired with an auxiliary array of DNA sequence objects in `seqArrayStack`. These objects are used to hold the interior nodes' state sets during the Fitch parsimony scoring algorithm.

```
      // Set up stack of auxiliary DNA sequence arrays.
      DnaSequence[][] seqArrayStack = new DnaSequence [N] [];
      for (int i = 0; i < N; ++ i)
         {
         DnaSequence[] seqArray = new DnaSequence [i];
         seqArrayStack[i] = seqArray;
```

```
            for (int j = 0; j < i; ++ j)
               {
               seqArray[j] = new DnaSequence (L);
               }
            }
```

The `signature` variable holds the signature of the tree under construction. `signature[0]` gives the position in `treeStack[0]` at which the first sequence was added, `signature[1]` gives the position in `treeStack[1]` at which the second sequence was added, and so on. The positions are all initialized to –1 because of the way the tree construction works, as will be seen.

```
         // Set up tree signature.
         int[] signature = new int [N];
         for (int i = 1; i < N; ++ i)
            {
            signature[i] = -1;
            }
```

Now comes the actual search. Conceptually, the program is searching a graph of all possible tree topologies (Figure 38.6). The search graph is organized into *N* levels. At level 0, each graph vertex corresponds to a topology with the first sequence added; at level 1, each vertex corresponds to a topology with the second sequence added; and so on. Each vertex at one level is joined to vertices at the next level corresponding to the different positions at which the next sequence can be added. The graph is searched in a depth-first fashion. The `level` variable keeps track of the search's current level. The `signature` variable keeps track of the position at which each sequence is added to the growing tree at the current level.

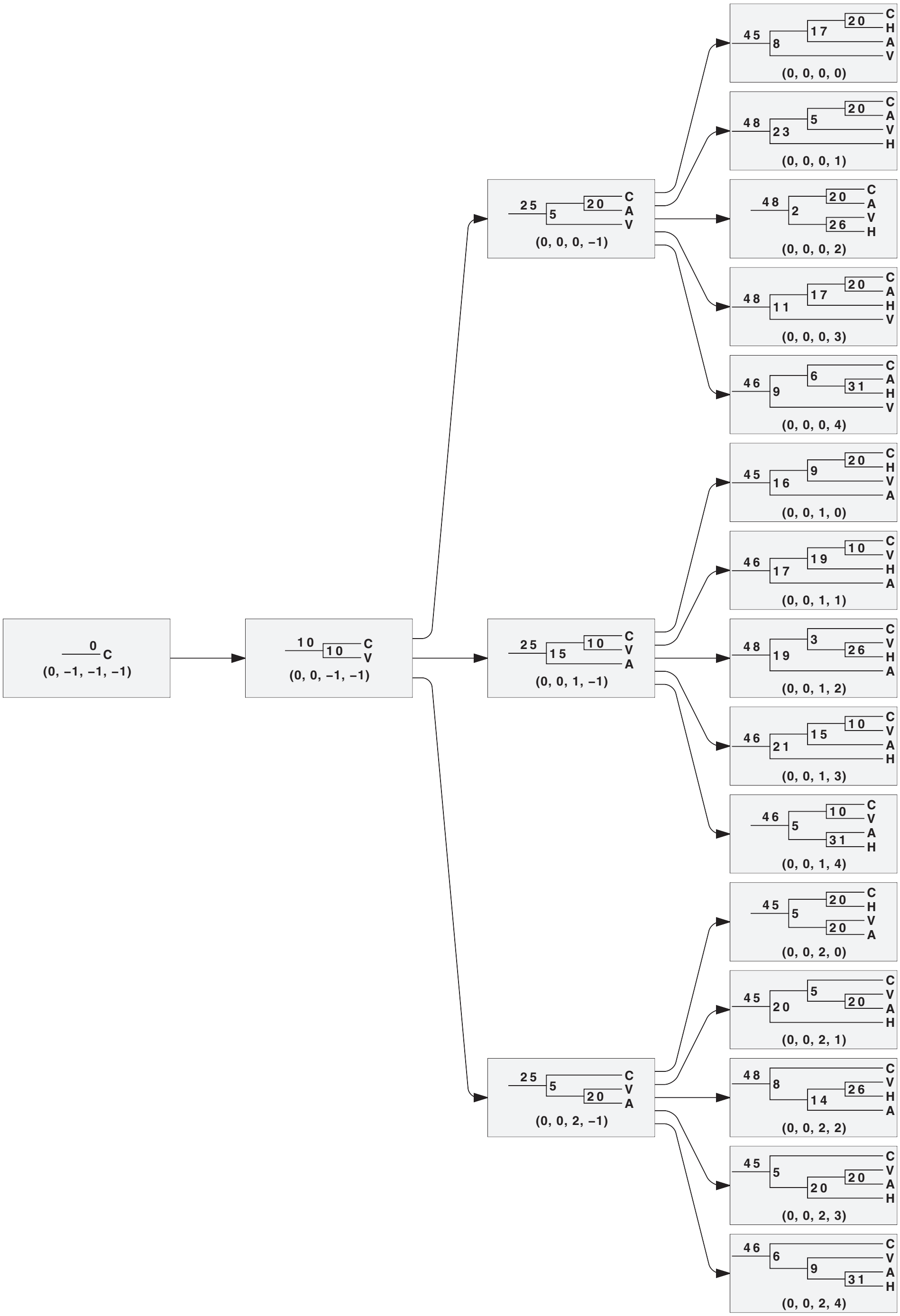


**Figure 38.6** Search graph for example sequences; each vertex shows tree, overall parsimony score, score at each interior node, and tree signature

```
// Initialize DNA sequence tree at level 0 of the search
// graph.
treeStack[0].add (0, seqList.seq(0));

// Traverse levels 1 .. N-1 of the search graph.
int level = 1;
while (level > 0)
   {
   DnaSequenceTree prevTree = treeStack[level-1];

   // If we have reached the bottom of the search graph, we
   // have a tentative solution.
   if (level == N)
      {
      int tentativeScore =
         prevTree.seq (prevTree.root()) .score();

      // If tentative solution’s score is better than the
      // best solution’s score, discard previous solutions.
      if (tentativeScore < score)
         {
         treeList.clear();
         score = tentativeScore;
         }

      // If tentative solution’s score is the same as the
      // best solution’s score, record tentative solution.
      if (tentativeScore == score &&
               treeList.size() < treeStoreLimit)
         {
         treeList.add ((int[]) signature.clone());
         }

      // Go to previous level.
      -- level;
      }

   // If there are no more positions to try at this level,
   // reset position at this level and go to previous level.
   else if (signature[level] == 2*(level - 1))
      {
      signature[level] = -1;
      -- level;
      }
```

```
            // If there are more positions to try at this level, add
            // the DNA sequence to the tree at the next position and
            // go to the next level.
            else
               {
               ++ signature[level];
               DnaSequenceTree currTree = treeStack[level];
               currTree.copy (prevTree);
               int tip = currTree.add
                  (signature[level], seqList.seq(level));
               FitchParsimony.updateScore
                  (currTree, tip, seqArrayStack[level]);
               ++ level;
               }
            }
         }
      }
```

We could have coded the depth-first graph search using recursion, with a search method calling itself recursively to go to the next level and returning to go to the previous level. Instead, we coded the depth-first graph search using iteration, with a while loop and a level variable incrementing and decrementing. This reduces the running time; it takes less time to go back to the top of a loop than to call a method. To reduce the running time further, we save the partially constructed trees in an explicit stack, so we can instantly backtrack to the tree at the previous level. To reduce the running time still further, we allocate all the necessary tree objects in advance rather than continually constructing new objects, which eliminates the time needed to execute constructors.

- Class Results has a method to output the results of the search. The results are stored in an HTML file and can be viewed with a Web browser. The results include the input DNA sequences; distance matrices of the Hamming distances and the Jukes-Cantor corrected distances; and the most parsimonious trees (those with the smallest parsimony score) displayed textually and graphically. Each tree is annotated with the least-squares branch lengths computed for that topology as well as the squared error in the branch lengths.
- Class HammingDistance, which implements interface Distance, computes the Hamming distance between two sequences.
- Class JukesCantorDistance, which also implements interface Distance, computes the Jukes-Cantor corrected distance between two sequences.
- Class LeastSquaresBranchLength solves Equation 38.4 to find the least-squares branch lengths for a given topology and sequences. It uses class edu.rit.numeric.NonNegativeLeastSquares to do the calculations.

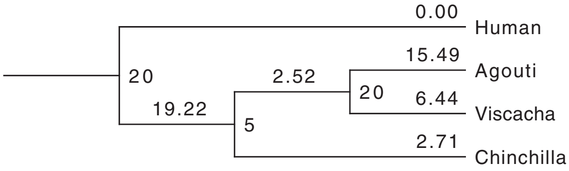
- Class edu.rit.draw.Drawing lets a program create drawings (like the figures in this chapter) and save them in files of various formats, including PostScript and PNG.
- Class TreeDrawing makes a drawing of a DNA sequence tree.

Finally, class PhylogenyParsExhSeq is the main program for sequential maximum parsimony phylogenetic tree construction. The command-line arguments are the following:

- The name of the input file containing the DNA sequences in PHYLIP format.
- The name of the directory in which to store the output HTML file containing the results, the PNG files containing the tree graphics, and other output files.
- *N*, the number of DNA sequences from the input file to include in the search. If *N* is specified, then only the first *N* sequences are included. If *N* is not specified, then all are included.
- *T*, the maximum number of phylogenetic trees to report. If not specified, then *T* = 100 is used. This is to prevent running out of memory if many topologies all have the same smallest parsimony score.

```
package edu.rit.compbio.phyl;
import edu.rit.pj.Comm;
import java.io.File;
import java.util.HashMap;
public class PhylogenyParsExhSeq
   {
   public static void main
      (String[] args)
      throws Exception
      {
      // Start timing.
      long t1 = System.currentTimeMillis();

      // Parse command line arguments.
      if (args.length < 2 || args.length > 4) usage();
      File infile = new File (args[0]);
      File outdir = new File (args[1]);
      int T = 100;
      if (args.length >= 4) T = Integer.parseInt (args[3]);

      // Read DNA sequence list from file and truncate to N
      // sequences if necessary.
      DnaSequenceList seqList = DnaSequenceList.read (infile);
      int N = seqList.length();
      if (args.length >= 3) N = Integer.parseInt (args[2]);
      seqList.truncate (N);
      // Excise uninformative sites.
      DnaSequenceList excisedList = new DnaSequenceList (seqList);
      int uninformativeScore =
         excisedList.exciseUninformativeSites();

      // Map each excised DNA sequence to its original DNA
      // sequence.
      HashMap<DnaSequence,DnaSequence> seqMap =
         new HashMap<DnaSequence,DnaSequence>();
      for (int i = 0; i < N; ++ i)
         {
         seqMap.put (excisedList.seq (i), seqList.seq (i));
         }

      long t2 = System.currentTimeMillis();

      // Run the exhaustive search.
      MaximumParsimony searcher = new MaximumParsimonyExhSeq();
      searcher.findTrees (excisedList, T, -1);
      searcher.score += uninformativeScore;

      long t3 = System.currentTimeMillis();

      // Report results.
      Results.report
         (/*directory   */ outdir,
          /*programName */ "PhylogenyParsExhSeq",
          /*hostName    */ Comm.world().host(),
          /*K           */ 1,
          /*infile      */ infile,
          /*seqList     */ seqList,
          /*seqMap      */ seqMap,
          /*searcher    */ searcher,
          /*t1          */ t1,
          /*t2          */ t2,
          /*t3          */ t3);

      // Stop timing.
      long t4 = System.currentTimeMillis();
      System.out.println ((t2-t1)+" msec pre");
      System.out.println ((t3-t2)+" msec calc");
      System.out.println ((t4-t3)+" msec post");
      System.out.println ((t4-t1)+" msec total");
      }
   }
```

Figure 38.6 shows the 15 trees the PhylogenyParsExhSeq program generates, along with their parsimony scores. There are 5 trees with the smallest parsimony score of 45, and these are the trees the program reports. Here's a closer look at one of them:



Each interior node is marked with the number of state changes, as computed by the Fitch parsimony scoring algorithm. Each branch is marked with the least-squares branch length computed for this topology. The squared error in the distances is 5.92. The other four trees are the same as this one, differing only in the placement of the root node.

Does this mean that viscachas and agoutis actually descended from a common ancestor, rather than viscachas and chinchillas? Not really. We have found the most parsimonious phylogeny based only on one gene, the insulin gene, in each species. If we computed the phylogeny based on more of the species' genes, or if we included more species, then we would probably get a different answer. However, the program's running time goes up as the double factorial of the number of species. If we want to analyze more than a handful of species, we have to use something other than exhaustive search.

## 38.6 Maximum Parsimony with Branch-and-Bound Search

Let's go back for a moment to the eight informative sites in the four example sequences:

```
Chinchilla AACTGTCT
Viscacha   AGACATAT
Agouti     CGATAAAC
Human      CACCGACC
```

Suppose we are proceeding through the depth-first search of the graph and have added the first species. Consider the first site, whose state is A. When we add the remaining three species, the first site is also going to have a state of C somewhere in the tree. Consequently, the number of state changes is going to increase by at least 1 at the first site. If we count the number of additional states from the remaining three species at each site and total up the counts, we find that the number of state changes must increase by at least 8 when we add the second, third, and fourth species (depending on the topology, it may increase more):

```
Chinchilla AACTGTCT
           11111111 ← 8 total
Viscacha   AGACATAT
Agouti     CGATAAAC
Human      CACCGACC
```

When we add the third and fourth species, the number of state changes must increase by at least 3:

```
Chinchilla AACTGTCT
Viscacha   AGACATAT
           1    1 1 ← 3 total
Agouti     CGATAAAC
Human      CACCGACC
```

And when we add the fourth species, the number of state changes does not increase further:

```
Chinchilla AACTGTCT
Viscacha   AGACATAT
Agouti     CGATAAAC
                    ← 0 total
Human      CACCGACC
```

We can use this information to reduce—possibly, *greatly* reduce—the number of vertices we have to search to find the most parsimonious topology.

Recall that as we traverse the search graph, we are keeping track of the smallest parsimony score found so far; call it *S*. Suppose we're at a vertex in the search graph where we've added a certain number of species, and the tree's partial parsimony score at that point is *X*. Suppose that when we add the remaining species, the number of state changes (and thus the parsimony score) will increase by at least *Y*. Finally, suppose *X*+*Y* is greater than *S*. In this situation, there is no point in continuing the search past the current vertex. Every complete tree past this point will end up with a parsimony score worse than the solution we've found so far. Instead, we can immediately go on to the next vertex at the current level of the search graph. The information about the additional number of state changes from the remaining species has let us effectively snip out a whole section of the search graph. This is called **pruning** the search.

A search algorithm that prunes in this manner is called a **branch-and-bound search** algorithm. As the algorithm traverses the *branches* of the search graph, it continually evaluates a lower *bound* on the score that would be obtained if the search continued all the way to the bottom. If this lower bound ever exceeds the best score so far, the search is pruned.

It's important to emphasize that a branch-and-bound search is guaranteed to find the truly best solution, just like an exhaustive search. The only difference is that the branch-and-bound search can take less time.

The closer to the beginning of the search at which pruning happens, the smaller the program's running time will be. Thus, we want to find a bound close to the final score quickly, so we can prune large sections of the search graph. Finding a good initial bound is especially important in a phylogenetic tree search because of the double-factorial explosion in the number of topologies the deeper we go into the search graph.

One way is to use a fast heuristic algorithm to find a topology, compute the resulting tree's parsimony score, and use that as the initial bound for the branch-and-bound search. We will use UPGMA to find this initial bound. While UPGMA may not find the true most parsimonious solution, we hope that UPGMA finds a solution close to the most parsimonious.

We can use the output of UPGMA in another way as well. UPGMA yields a branch length for each tip node. Suppose we sort the list of species into descending order of this branch length. Then, as we add the species to the tree in this order, the species that are more distant from the others are added first. This tends to make the parsimony score increase more quickly at the initial levels of the search graph and increases the likelihood that the search gets pruned early on, thus reducing the running time to a greater extent.

The strategy of using the number of additional state changes as each remaining species is added was suggested in a 1979 paper by Foulds, Hendy, and Penny. Adding the most distant species first was suggested in a 1982 paper by Hendy and Penny. Using UPGMA to find an initial bound was suggested by O'Brien in 2006.

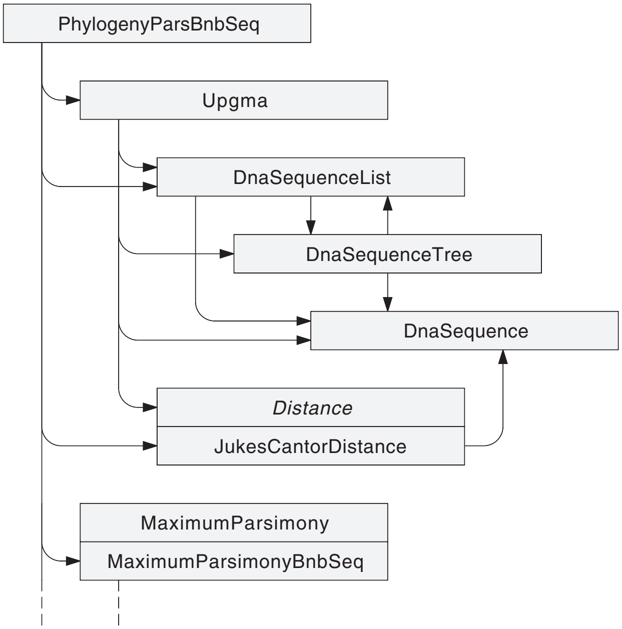


**Figure 38.7** PhylogenyParsBnbSeq class relationships (remainder is the same as PhylogenyParsExhSeq in Figure 38.5)

Pulling these ideas together, we can design a phylogenetic tree construction program using branch-and-bound search. It is still a sequential program (the next version will be a parallel program). Figure 38.7 shows the classes from which the branch-and-bound phylogeny program is built and their "uses" relationships. Most of the classes are the same as in the exhaustive search program. Here are the new classes:

- Class Upgma constructs a phylogenetic tree from a DNA sequence list using the UPGMA algorithm.
- Class MaximumParsimonyBnbSeq extends base class MaximumParsimony with the algorithm for a branch-and-bound search executed sequentially in a single thread. Most of it is the same as class MaximumParsimonyExhSeq for an exhaustive search.

```
package edu.rit.compbio.phyl;
import java.util.ArrayList;
public class MaximumParsimonyBnbSeq
   extends MaximumParsimony
```

```
{
/**
 * Construct a new maximum parsimony phylogenetic tree
 * construction algorithm object.
 */
public MaximumParsimonyBnbSeq()
   {
   }

/**
 * Find the maximum parsimony phylogenetic tree(s) for the given
 * DNA sequence list. The results are stored in the fields of
 * this object. The findTrees() method will store at most
 * treeStoreLimit maximum parsimony phylogenetic trees. The
 * findTrees() method will only find trees whose parsimony scores
 * are less than or equal to the initialBound.
 *
 * @param  seqList         DNA sequence list.
 * @param  treeStoreLimit  Maximum number of trees to store.
 * @param  initialBound    Initial bound for branch-and-bound
 *                         search.
 */
public void findTrees
   (DnaSequenceList seqList,
    int treeStoreLimit,
    int initialBound)
   {
   // Initialize.
   this.seqList = seqList;
   this.treeStoreLimit = treeStoreLimit;
   this.initialBound = initialBound;
   this.treeList = new ArrayList<int[]> (treeStoreLimit);
   this.score = initialBound;
   int L = seqList.seq(0).length();
   int N = seqList.length();
   int C = 2*N - 1;
```

Here, we assemble the information used later to compute the bound and prune the search.

```
   // Compute number of absent states as each DNA sequence is
   // added.
   int[] absentStates = seqList.countAbsentStates();
   // Set up stack of DNA sequence trees.
   DnaSequenceTree[] treeStack = new DnaSequenceTree [N];
   for (int i = 0; i < N; ++ i)
```

```
            {
            treeStack[i] = new DnaSequenceTree (C);
            }

         // Set up stack of auxiliary DNA sequence arrays.
         DnaSequence[][] seqArrayStack = new DnaSequence [N] [];
         for (int i = 0; i < N; ++ i)
            {
            DnaSequence[] seqArray = new DnaSequence [i];
            seqArrayStack[i] = seqArray;
            for (int j = 0; j < i; ++ j)
               {
               seqArray[j] = new DnaSequence (L);
               }
            }

         // Set up tree signature.
         int[] signature = new int [N];
         for (int i = 1; i < N; ++ i)
            {
            signature[i] = -1;
            }
```

Here is the actual branch-and-bound search. It is the same as the exhaustive search, except for one addition—pruning.

```
         // Initialize DNA sequence tree at level 0 of the search
         // graph.
         treeStack[0].add (0, seqList.seq(0));

         // Traverse levels 1 .. N-1 of the search graph.
         int level = 1;
         while (level > 0)
            {
            DnaSequenceTree prevTree = treeStack[level-1];

            // If we have reached the bottom of the search graph, we
            // have a tentative solution.
            if (level == N)
               {
               int tentativeScore =
                  prevTree.seq (prevTree.root()) .score();

               // If tentative solution's score is better than the
               // best solution's score, discard previous solutions.
```

```
   if (tentativeScore < score)
      {
      treeList.clear();
      score = tentativeScore;
      }

   // If tentative solution’s score is the same as the
   // best solution’s score, record tentative solution.
   if (tentativeScore == score &&
            treeList.size() < treeStoreLimit)
      {
      treeList.add ((int[]) signature.clone());
      }

   // Go to previous level.
   -- level;
   }

// If there are no more positions to try at this level,
// reset position at this level and go to previous level.
else if (signature[level] == 2*(level - 1))
   {
   signature[level] = -1;
   -- level;
   }

// If there are more positions to try at this level, add
// the DNA sequence to the tree at the next position and
// do branch-and-bound.
else
   {
   ++ signature[level];
   DnaSequenceTree currTree = treeStack[level];
   currTree.copy (prevTree);
   int tip = currTree.add
      (signature[level], seqList.seq(level));
   int partialScore =
      FitchParsimony.updateScore
         (currTree, tip, seqArrayStack[level]);
```

Here is where the pruning happens. Instead of going to the next level unconditionally, we compute the bound and go to the next level only if the bound does not exceed the best score found so far.

```
            // If partial parsimony score plus number of absent
            // states in the remaining levels is less than or equal
            // to the best solution's score, go to the next level,
            // otherwise try the next choice at this level.
            if (partialScore + absentStates[level] <= score)
               {
               ++ level;
               }
            }
         }
      }
   }
```

- Class PhylogenyParsBnbSeq is the main program for sequential branch-and-bound maximum parsimony phylogenetic tree construction. The command-line arguments are the same as the PhylogenyParsExhSeq program.

```
package edu.rit.compbio.phyl;
import edu.rit.pj.Comm;
import java.io.File;
import java.util.HashMap;
public class PhylogenyParsBnbSeq
   {
   public static void main
      (String[] args)
      throws Exception
      {
      // Start timing.
      long t1 = System.currentTimeMillis();

      // Parse command line arguments.
      if (args.length < 2 || args.length > 4) usage();
      File infile = new File (args[0]);
      File outdir = new File (args[1]);
      int T = 100;
      if (args.length >= 4) T = Integer.parseInt (args[3]);

      // Read DNA sequence list from file and truncate to N
      // sequences if necessary.
      DnaSequenceList seqList = DnaSequenceList.read (infile);
      int N = seqList.length();
      if (args.length >= 3) N = Integer.parseInt (args[2]);
      seqList.truncate (N);
```

Before commencing the branch-and-bound search, we run the fast heuristic UPGMA algorithm to get a tentative topology.

```
// Run the UPGMA algorithm to get an approximate solution.
// Calculate its parsimony score.
DnaSequenceTree upgmaTree =
   Upgma.buildTree (seqList, new JukesCantorDistance());
int upgmaScore = FitchParsimony.computeScore (upgmaTree);
```

The following ensures that during branch-and-bound search, the sequences most distant from the others are added first.

```
// Put the DNA sequence list in descending order of tip node
// branch length in the UPGMA tree.
DnaSequenceList sortedList = upgmaTree.toList();

// Excise uninformative sites.
DnaSequenceList excisedList =
   new DnaSequenceList (sortedList);
int uninformativeScore =
   excisedList.exciseUninformativeSites();

// Map each excised DNA sequence to its original DNA
// sequence.
HashMap<DnaSequence,DnaSequence> seqMap =
   new HashMap<DnaSequence,DnaSequence>();
for (int i = 0; i < N; ++ i)
   {
   seqMap.put (excisedList.seq (i), sortedList.seq (i));
   }

long t2 = System.currentTimeMillis();
```

Here's where we use the UPGMA tree's parsimony score as the initial bound in the branch-and-bound search.

```
// Run the branch-and-bound search. Use the UPGMA parsimony
// score (reduced by score from uninformative sites) as the
// initial bound.
MaximumParsimony searcher = new MaximumParsimonyBnbSeq();
searcher.findTrees
   (excisedList, T, upgmaScore - uninformativeScore);
searcher.initialBound += uninformativeScore;
searcher.score += uninformativeScore;
```

```
      long t3 = System.currentTimeMillis();

      // Report results.
      Results.report
         (/*directory   */ outdir,
          /*programName */ "PhylogenyParsBnbSeq",
          /*hostName    */ Comm.world().host(),
          /*K           */ 1,
          /*infile      */ infile,
          /*seqList     */ seqList,
          /*seqMap      */ seqMap,
          /*searcher    */ searcher,
          /*t1          */ t1,
          /*t2          */ t2,
          /*t3          */ t3);

      // Stop timing.
      long t4 = System.currentTimeMillis();
      System.out.println ((t2-t1)+" msec pre");
      System.out.println ((t3-t2)+" msec calc");
      System.out.println ((t4-t3)+" msec post");
      System.out.println ((t4-t1)+" msec total");
      }
   }
```

How much faster than the exhaustive search is the branch-and-bound search? The PhylogenyParsExhSeq and PhylogenyParsBnbSeq programs were run on one CPU of the “parasite” SMP parallel computer. The input was a group of 18 DNA sequences from 18 iguana species. Each sequence was 900 characters long. Both programs were run with $N = 9$, 10, and 11 sequences. The branch-and-bound program was also run with $N = 12$, 13, 14, and 15 sequences. The running times (msec) for the search section of each program were the following:

| *N* | Exhaustive | Branch-and-bound |
|---:|---:|---:|
| 9 | 5451 | 187 |
| 10 | 111342 | 858 |
| 11 | 2571056 | 1406 |
| 12 | *53992176* | 7667 |
| 13 | *1241820048* | 32240 |
| 14 | *31045501200* | 117484 |
| 15 | *838228532400* | 1455052 |

The exhaustive program's running times for $N = 12$ to 15 were estimated assuming the running time is proportional to $(2N - 3)!!$. The advantage of branch-and-bound search is clear. Yet even branch-and-bound search can benefit from an additional speedup on a parallel computer as the problem size $N$ continues to increase.

## 38.7 Parallel Branch-and-Bound Search

At last, we can design an SMP parallel branch-and-bound maximum parsimony phylogenetic tree construction program, based on the sequential version. The first design decision is how to partition the computation among the parallel threads. The job is a bit more complicated than dividing a for loop index range into chunks, as we have done with most of our parallel programs. Instead, we have to divide the *vertices of the search graph* into chunks (Figure 38.8). Yet we still want to use the *parallel for loop* pattern, which requires an index range. Here's how we'll do it. We'll partition the vertices at level 6 of the search graph, the level at which we add the seventh species to the tree. (Levels are numbered starting from 0.) There are $(2 \cdot 7 - 3)!! = 10{,}395$ vertices at this level. So, we'll let our parallel for loop index run from 0 to 10,394. Each thread constructs the tree topology at level 6 corresponding to the lower bound of the thread's chunk of loop indexes, and then commences the search from there. The choice of the level at which to partition the search graph is more or less arbitrary; level 7 (with 114,345 vertices) or level 8 (with 1,486,485 vertices) would also work.

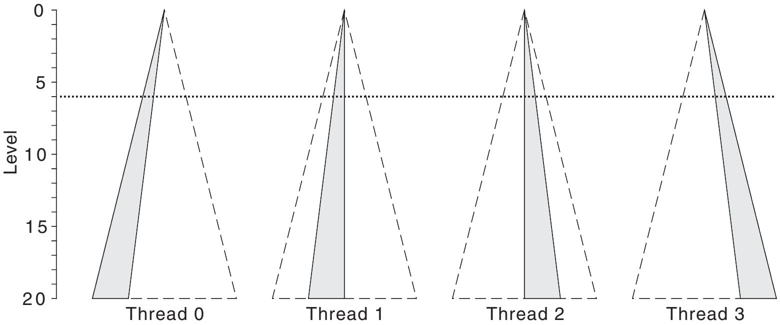


**Figure 38.8** Search graph partitioned among parallel threads

As usual, the parallel for loop divides the index range into chunks according to the loop schedule. Because a branch-and-bound algorithm prunes the search at various levels, depending on the parsimony scores encountered, partitioning the search graph equally among the threads—a fixed schedule—will likely yield an unbalanced load. Instead, a dynamic schedule or guided schedule must be used to balance the load.

The second design decision concerns shared variables. The two key variables the search algorithm writes are `treeList`, the list of most parsimonious trees (actually, tree signatures), and `score`, the smallest parsimony `score`. These are fields of the base class MaximumParsimony. For the tree list, we'll use the *reduction* pattern. The `treeList` field becomes a shared global variable. Each thread has its own per-thread tree list in which the thread stores the most parsimonious trees the thread finds in its own chunks of the search graph, without needing to synchronize with the other threads. Each thread's last act is to copy the trees from its own list to the global tree list, and when doing so, the thread must synchronize with the other threads to avoid conflicts while updating the shared variable.

The smallest parsimony score variable needs to be treated a little differently. Whenever one thread finds a tree with a smaller score than the current score, *all* the threads need to know about the new smaller score immediately, so all the threads can prune their searches as soon as possible. Therefore, we can't use the reduction pattern for the `score` variable. The threads must all access a *shared* variable rather than per-thread variables. Whenever a thread reads the shared variable to compute the bound for pruning, and whenever a thread updates the shared variable with a new smaller score, the thread must synchronize with the other threads to prevent conflicts. To achieve synchronization, we will make the shared variable an instance of class edu.rit.pj.reduction.SharedInteger, the multiple-thread-safe integer wrapper class.

Figure 38.9 shows the classes from which the SMP parallel branch-and-bound phylogeny program is built and their "uses" relationships. Most of the classes are the same as in the exhaustive search program and the sequential branch-and-bound search program.

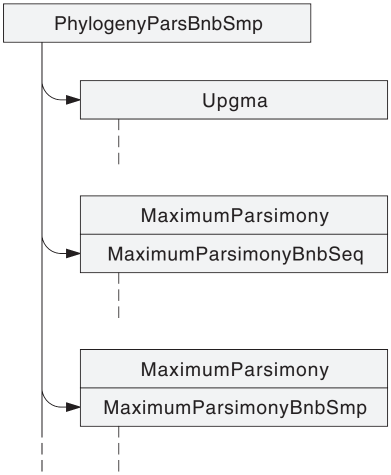


**Figure 38.9** PhylogenyParsBnbSmp class relationships (remainder is the same as PhylogenyParsExhSeq in Figure 38.5 and PhylogenyParsBnbSeq in Figure 38.7)

The main program is class PhylogenyParsBnbSmp. It differs from the sequential version in only one place. Here is the sequential version.

```
MaximumParsimony searcher = new MaximumParsimonyBnbSeq();
searcher.findTrees
   (excisedList, T, upgmaScore - uninformativeScore);
```

Alternatively, here is the SMP parallel version.

```
ParallelTeam team = null;
MaximumParsimony searcher = null;
if (seqList.length() > 7)
   {
   team = new ParallelTeam();
```

```
         searcher = new MaximumParsimonyBnbSmp (team);
         }
      else
         {
         searcher = new MaximumParsimonyBnbSeq();
         }
      searcher.findTrees
         (excisedList, T, upgmaScore - uninformativeScore);
```

To do the search, the parallel version uses an instance of class MaximumParsimonyBnbSmp with a parallel thread team; except if there are seven or fewer sequences, the search is done in a single thread. (The search only takes a fraction of a second in this case.)

Class MaximumParsimonyBnbSmp extends base class MaximumParsimony with the algorithm for a branch-and-bound search executed in parallel by a team of threads. Most of it is the same as the sequential version, class MaximumParsimonyBnbSeq.

```
package edu.rit.compbio.phyl;
import edu.rit.pj.IntegerForLoop;
import edu.rit.pj.IntegerSchedule;
import edu.rit.pj.ParallelRegion;
import edu.rit.pj.ParallelSection;
import edu.rit.pj.ParallelTeam;
import edu.rit.pj.reduction.IntegerOp;
import edu.rit.pj.reduction.SharedInteger;
import java.util.ArrayList;
public class MaximumParsimonyBnbSmp
   extends MaximumParsimony
   {
   private ParallelTeam team;

   /**
    * Construct a new maximum parsimony phylogenetic tree
    * construction algorithm object.
    *
    * @param  team  Parallel thread team that will do the search.
    */
   public MaximumParsimonyBnbSmp
      (ParallelTeam team)
      {
      if (team == null)
         {
         throw new NullPointerException
            ("MaximumParsimonyBnbSmp(): team is null");
         }
      this.team = team;
      }

   /**
    * Find the maximum parsimony phylogenetic tree(s) for the given
    * DNA sequence list. The results are stored in the fields of
    * this object. The findTrees() method will store at most
    * treeStoreLimit maximum parsimony phylogenetic trees. The
    * findTrees() method will only find trees whose parsimony scores
    * are less than or equal to the initialBound.
    *
    * The findTrees() method assumes there are more than 7 DNA
    * sequences. If there are 7 or fewer DNA sequences, don't
    * bother computing trees in parallel, use class
    * MaximumParsimonyBnbSeq instead.
    *
    * @param  seqList         DNA sequence list.
    * @param  treeStoreLimit  Maximum number of trees to store.
    * @param  initialBound    Initial bound for branch-and-bound
    *                         search.
    */
   public void findTrees
      (final DnaSequenceList seqList,
       final int treeStoreLimit,
       final int initialBound)
      throws Exception
      {
      // Initialize.
      this.seqList = seqList;
      this.treeStoreLimit = treeStoreLimit;
      this.initialBound = initialBound;
      this.treeList = new ArrayList<int[]> (treeStoreLimit);
      this.score = initialBound;
      final int L = seqList.seq(0).length();
      final int N = seqList.length();
      final int C = 2*N - 1;

      // Compute number of absent states as each DNA sequence is
      // added.
      final int[] absentStates = seqList.countAbsentStates();
```

Here is the multiple-thread-safe shared variable, `bound`, that holds the bound (smallest parsimony score) that all the threads will use to prune their searches.

```
// Shared bound for branch-and-bound, set to initial parsimony
// score.
final SharedInteger bound = new SharedInteger (score);

// Begin parallel execution.
team.execute (new ParallelRegion()
   {
   public void run() throws Exception
      {
      // Do the 10,395 alternatives at level 6 (seventh
      // level) in parallel.
      execute (0, 10394, new IntegerForLoop()
         {
         // Thread local variables.
         int thrScore;
         ArrayList<int[]> thrTreeList;
         DnaSequenceTree[] treeStack;
         DnaSequence[][] seqArrayStack;
         int[] signature;

         // Extra padding to avert cache interference.
         long p0, p1, p2, p3, p4, p5, p6, p7;
         long p8, p9, pa, pb, pc, pd, pe, pf;

         // Initialize thread local variables.
         public void start()
            {
            // Set up list of maximum parsimony tree
            // signatures.
            thrScore = score;
            thrTreeList =
               new ArrayList<int[]> (treeStoreLimit);

            // Set up stack of DNA sequence trees.
            treeStack = new DnaSequenceTree [N];
            for (int i = 0; i < N; ++ i)
               {
               treeStack[i] = new DnaSequenceTree (C);
               }

            // Set up stack of auxiliary DNA sequence arrays.
            seqArrayStack = new DnaSequence [N] [];
            for (int i = 0; i < N; ++ i)
               {
               DnaSequence[] seqArray = new DnaSequence [i];
               seqArrayStack[i] = seqArray;
               for (int j = 0; j < i; ++ j)
                  {
                  seqArray[j] = new DnaSequence (L);
                  }
               }

            // Set up tree signature.
            signature = new int [N];

            // Initialize DNA sequence tree at level 0.
            treeStack[0].add (0, seqList.seq(0));
            }
```

The parallel for loop will use the schedule specified on the command line with the `-Dpj.schedule` flag. If the schedule is not specified, the default is a dynamic schedule with a chunk size of 100, for load balancing.

```
// Specify parallel loop schedule.
public IntegerSchedule schedule()
   {
   return IntegerSchedule.runtime
      (IntegerSchedule.dynamic (100));
   }
```

Here is the parallel for loop body. The lower-bound index, `first`, designates the level-6 tree topology at which to start searching. The index is used to initialize the tree signature as follows. Because there are 11 alternatives at level 6, we divide the index by 11; the remainder is `signature[6]`, the quotient replaces the index. Because there are 9 alternatives at level 5, we divide the index by 9; the remainder is `signature[5]`, the quotient replaces the index. Continuing this way, we fill out `signature[6]` through `signature[1]`. `Signature[0]` and `signature[7]` and above are all set to zero. For example, if first is 10240, then the signature through level 6 is (0, 0, 2, 4, 5, 3, 10):

10240 ÷ 11 = 930 remainder 10
930 ÷ 9 = 103 remainder 3
103 ÷ 7 = 14 remainder 5
14 ÷ 5 = 2 remainder 4
2 ÷ 3 = 0 remainder 2
0 ÷ 1 = 0 remainder 0

Actually, the elements of `signature` are set to one less than the preceding because of the way the graph search loop works.

```
// Do a chunk of iterations.
public void run (int first, int last)
   {
```

```
// Initialize signature as specified by lower
// bound loop index <first>.
int q = first;
for (int i = 6; i > 0; — i)
   {
   int d = 2*i - 1;
   signature[i] = q % d - 1;
   q = q / d;
   }
for (int i = 7; i < N; ++ i)
   {
   signature[i] = -1;
   }

// Traverse levels 1 .. N-1 of the search graph.
int level = 1;
while (level > 0)
   {
   DnaSequenceTree prevTree = treeStack[level-1];

   // If we have reached the bottom of the search
   // graph, we have a tentative solution.
   if (level == N)
      {
      int tentativeScore =
         prevTree.seq (prevTree.root()) .score();
```

Here, the thread updates the shared bound variable with the smaller of the current bound and the tentative solution’s score. The thread synchronizes with the other threads accessing bound by calling the multiple-thread-safe `reduce()` method to do the update. This method computes the minimum (using the reduction operator `IntegerOp.MINIMUM`) of `bound` and `tentativeScore`, stores the result back into `bound`, and returns the result. (See Appendix D for further information about what happens under the hood in the `reduce()` method.)

```
      // Atomically set global bound to the
      // smaller of global bound and tentative
      // score.
      int newBound = bound.reduce
         (tentativeScore, IntegerOp.MINIMUM);
      // If global bound is less than previous
      // best solution’s score, discard previous
      // solutions.
      if (newBound < thrScore)
         {
```

```
      thrScore = newBound;
      thrTreeList.clear();
      }

   // If tentative solution’s score is the
   // same as best solution’s score, record
   // solution.
   if (tentativeScore == thrScore &&
         thrTreeList.size() < treeStoreLimit)
      {
      thrTreeList.add
         ((int[]) signature.clone());
      }

   // Go to previous level.
   -- level;
   }

// If there are no more positions to try at
// this level, reset position at this level
// and go to previous level.
else if (signature[level] == 2*(level - 1))
   {
   signature[level] = -1;
   -- level;
   }

// If there are more positions to try at this
// level, add the DNA sequence to the tree at
// the next position and do branch-and-bound.
else
   {
   ++ signature[level];
   DnaSequenceTree currTree =
      treeStack[level];
   currTree.copy (prevTree);
   int tip =
      currTree.add
         (signature[level],
          seqList.seq(level));
   int partialScore =
      FitchParsimony.updateScore
         (currTree,
          tip,
          seqArrayStack[level]);
```

The logic for pruning the search is slightly different in the parallel version. We won't prune at all if we're below level 6. This is to make sure the parallel for loop visits all the search graph vertices at level 6.

```
      // If we're below level 6, branch.
      if (level < 6)
         {
         ++ level;
         }
```

If we're at level 6, we won't prune, and we will also count off the number of level-6 vertices we've visited. When we've visited all the vertices in the chunk from index `first` to index `last`, we return from the parallel for loop's `run()` method.

```
      // If we're at level 6 and we're not done
      // with this chunk of iterations, advance
      // to next iteration and branch.
      else if (level == 6 && first <= last)
         {
         ++ first;
         ++ level;
         }

      // If we're at level 6 and we are done with
      // this chunk of iterations, stop this
      // chunk.
      else if (level == 6) return;
```

Above level 6, we'll prune the search as we did in the sequential version. To compute the bound for the pruning decision, we call the multiple-thread-safe `get()` method on the shared `bound` variable, thus synchronizing with other threads reading or updating the variable.

```
      // We're above level 6. If partial
      // parsimony score plus number of absent
      // states in the remaining levels is less
      // than or equal to the best solution's
      // score, go to the next level, otherwise
      // try the next choice at this level.
      else if (partialScore + absentStates[level]
               <= bound.get())
         {
         ++ level;
         }
      }
   }
}
```

Here is the thread's last act—reducing the per-thread tree list into the shared tree list. Synchronization is achieved by putting the code in a critical section.

```
            // Reduce per-thread list of solutions into global
            // list of solutions.
            public void finish() throws Exception
               {
               region().critical (new ParallelSection()
                  {
                  public void run()
                     {
                     // If this thread's best score is the same
                     // as global best score, add this thread's
                     // solutions to global list.
                     if (thrScore == bound.get())
                        {
                        for
                           (int i = 0;
                            i < thrTreeList.size() &&
                              treeList.size() < treeStoreLimit;
                            ++ i)
                           {
                           treeList.add (thrTreeList.get (i));
                           }
                        }
                     }
                  });
               }
            });
         }
      });

   // Finally, record global best score.
   score = bound.get();
   }
}
```

Table 38.1 (at the end of the chapter) lists, and Figure 38.10 plots, the PhylogenyParsBnbSmp program's performance on the "parasite" SMP parallel computer. The input was a group of 18 DNA sequences from 18 iguana species. Each sequence was 900 characters long. The program was run with $N = 13$, 14, and 15 sequences. The running-time metrics are for the search section of the program. The parallel program achieved efficiencies of 87 percent or better, out to eight processors.

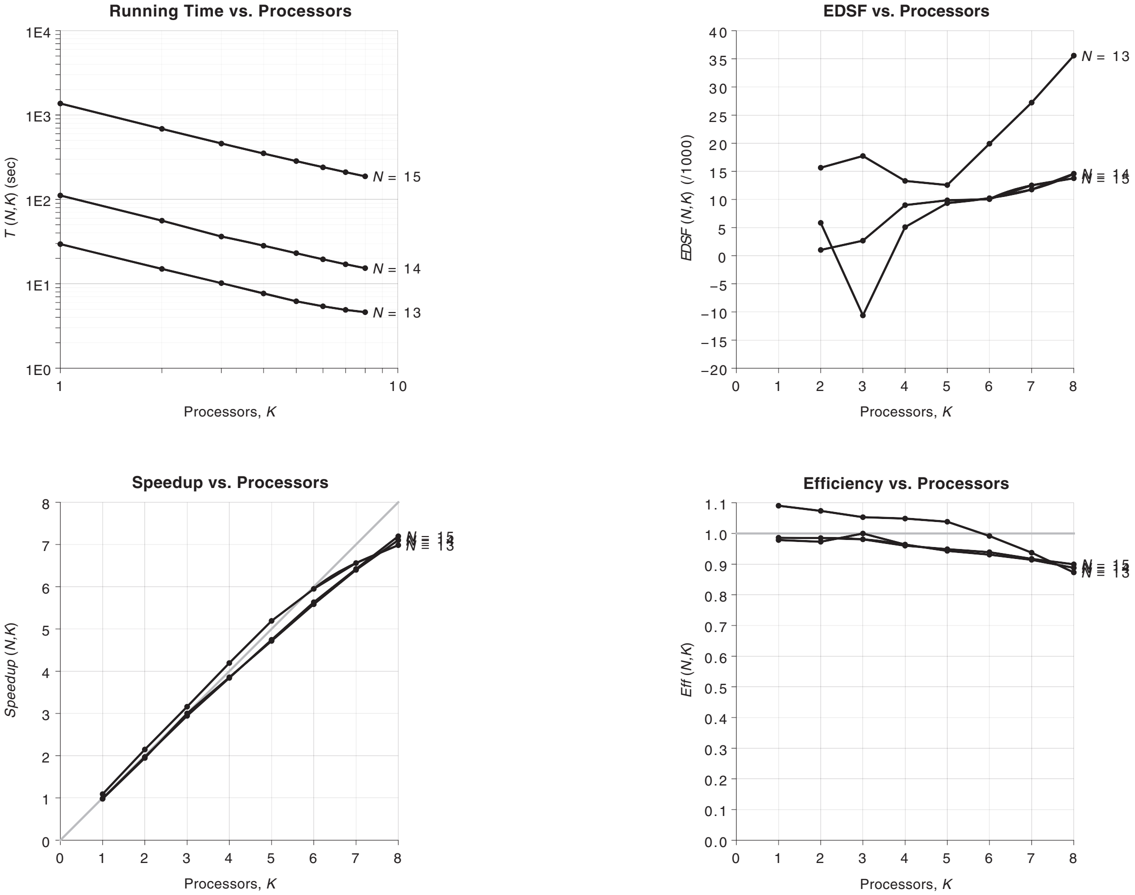


**Figure 38.10** PhylogenyParsBnbSeq/Smp running-time metrics

Going from an exhaustive search to a branch-and-bound search increased the number of species the maximum parsimony phylogenetic tree construction program could compute in a reasonable amount of time. Going to a parallel branch-and-bound search increased the number of species still further. However, the double-factorial explosion in the size of the search graph will inevitably overwhelm even the largest parallel supercomputer as the number of species increases. Finding the exact most parsimonious phylogenetic tree by branch-and-bound search is doable only for a few tens of species at most. Beyond that, branch-and-bound search takes too long—even on a parallel computer—and heuristic algorithms are the only viable alternative.

## 38.8 Acknowledgments

I am indebted to Joseph Felsenstein at the University of Washington and his monograph, *Inferring Phylogenies*, for the models and algorithms described in this chapter and their references. Larry Buckley of the Department of Biological Sciences at the Rochester Institute of Technology provided the iguana

DNA sequences used in the running-time measurements. For his M.S. degree project, my student Terence O'Brien developed a cluster parallel branch-and-bound phylogenetic tree construction program using C and MPI; I have used some of his ideas in my branch-and-bound programs.

## 38.9 For Further Information

On the discovery of the DNA double helix—the scientific papers:

- J. Watson and F. Crick. A structure for deoxyribose nucleic acid. *Nature*, 171:737–738, April 25, 1953.
- M. Wilkins, A. Stokes, and H. Wilson. Molecular structure of deoxypentose nucleic acids. *Nature*, 171:738–740, April 25, 1953.
- R. Franklin and R. Gosling. Molecular configuration in sodium thymonucleate. *Nature*, 171:740–741, April 25, 1953.

On the people who discovered the DNA double helix:

- J. Watson. *The Double Helix*. Atheneum Publishers, 1968.
- F. Crick. *What Mad Pursuit: A Personal View of Scientific Discovery*. Basic Books, 1990.
- B. Maddox. *Rosalind Franklin: The Dark Lady of DNA*. HarperCollins Publishers, 2002.
- M. Wilkins. *The Third Man of the Double Helix*. Oxford University Press, 2003.

On phylogenies and methods for inferring them:

- J. Felsenstein. *Inferring Phylogenies*. Sinauer Associates, 2004.

On PHYLIP:

- J. Felsenstein. PHYLIP programs and documentation. http://evolution.genetics.washington.edu/phylip/phylip.html

On the Jukes-Cantor model of DNA state changes:

- T. Jukes and C. Cantor. Evolution of protein molecules. In M. Munro, editor. *Mammalian Protein Metabolism, Volume III*. Academic Press, 1969, pages 21–132.
- J. Felsenstein. *Inferring Phylogenies*. Sinauer Associates, 2004, pages 156–158.

On UPGMA:

- R. Sokal and C. Michener. A statistical method for evaluating systematic relationships. *University of Kansas Science Bulletin*, 38:1409–1438, 1958.
- J. Felsenstein. *Inferring Phylogenies*. Sinauer Associates, 2004, pages 161–166.

On the Fitch algorithm for computing the parsimony score:

- W. Fitch. Toward defining the course of evolution: minimum change for a specified tree topology. *Systematic Zoology*, 20:406–416, 1971.
- J. Felsenstein. *Inferring Phylogenies*. Sinauer Associates, 2004, pages 11–13.

On using informative sites to reduce the time to compute the parsimony score:

- V. Ratner, A. Zharkikh, N. Kolchanov, S. Rodin, V. Solovyov, and A. Antonov. *Molecular Evolution*. Springer-Verlag, 1996.
- J. Felsenstein. *Inferring Phylogenies*. Sinauer Associates, 2004, pages 65–66.

On strategies for conducting a branch-and-bound phylogenetic tree search:

- L. Foulds, M. Hendy, and D. Penny. A graph theoretic approach to the development of minimal phylogenetic trees. *Journal of Molecular Evolution*, 13:127–149, 1979.
- M. Hendy and D. Penny. Branch and bound algorithms to determine minimal evolutionary trees. *Mathematical Biosciences*, 59:277–290, 1982.
- P. Purdom, P. Bradford, K. Tamura, and S. Kumar. Single column discrepancy and dynamic max-mini optimizations for quickly finding the most parsimonious evolutionary trees. *Bioinformatics*, 16:140–151, 2000.
- J. Felsenstein. *Inferring Phylogenies*. Sinauer Associates, 2004, pages 64–65.
- T. O'Brien. Speedup of parsimonious phylogenetic tree evaluation via parallel branch and bound. Rochester Institute of Technology Department of Computer Science M.S. project, September 2006.
  http://www.cs.rit.edu:8080/ms/static/ark/2006/1/two6384/

On parallel programs for maximum parsimony phylogenetic tree construction:

- Q. Snell, M. Whiting, M. Clement, and D. McLaughlin. Parallel phylogenetic inference. In *Proceedings of the ACM/IEEE 2000 Conference on Supercomputing*, November 2000, page 35.
- T. O'Brien. Speedup of parsimonious phylogenetic tree evaluation via parallel branch and bound. Rochester Institute of Technology Department of Computer Science M.S. project, September 2006.
  http://www.cs.rit.edu:8080/ms/static/ark/2006/1/two6384/

**Table 38.1** PhylogenyParsBnbSeq/Smp running-time metrics

| *N* | *K* | *T* | *Spdup* | *Eff* | *EDSF* |
|---:|---:|---:|---:|---:|---:|
| 13 | seq | 32195 | | | |
| 13 | 1 | 29526 | 1.090 | 1.090 | |
| 13 | 2 | 14994 | 2.147 | 1.074 | 0.016 |
| 13 | 3 | 10191 | 3.159 | 1.053 | 0.018 |
| 13 | 4 | 7676 | 4.194 | 1.049 | 0.013 |
| 13 | 5 | 6202 | 5.191 | 1.038 | 0.013 |
| 13 | 6 | 5411 | 5.950 | 0.992 | 0.020 |
| 13 | 7 | 4907 | 6.561 | 0.937 | 0.027 |
| 13 | 8 | 4610 | 6.984 | 0.873 | 0.036 |
| 14 | seq | 108894 | | | |
| 14 | 1 | 111278 | 0.979 | 0.979 | |
| 14 | 2 | 55964 | 1.946 | 0.973 | 0.006 |
| 14 | 3 | 36305 | 2.999 | 1.000 | -0.011 |
| 14 | 4 | 28244 | 3.855 | 0.964 | 0.005 |
| 14 | 5 | 23088 | 4.716 | 0.943 | 0.009 |
| 14 | 6 | 19495 | 5.586 | 0.931 | 0.010 |
| 14 | 7 | 17019 | 6.398 | 0.914 | 0.012 |
| 14 | 8 | 15330 | 7.103 | 0.888 | 0.015 |
| 15 | seq | 1349625 | | | |
| 15 | 1 | 1368661 | 0.986 | 0.986 | |
| 15 | 2 | 685031 | 1.970 | 0.985 | 0.001 |
| 15 | 3 | 458661 | 2.943 | 0.981 | 0.003 |
| 15 | 4 | 351399 | 3.841 | 0.960 | 0.009 |
| 15 | 5 | 284522 | 4.743 | 0.949 | 0.010 |
| 15 | 6 | 239571 | 5.634 | 0.939 | 0.010 |
| 15 | 7 | 210195 | 6.421 | 0.917 | 0.013 |
| 15 | 8 | 187574 | 7.195 | 0.899 | 0.014 |

# APPENDIX A

# OpenMP

in which we take a quick look at SMP parallel programming with OpenMP; and we compare OpenMP's features and performance to Parallel Java

# APPENDIX A OpenMP

## A.1 OpenMP Programming

OpenMP is a standard application programming interface (API) for writing thread-based shared memory parallel programs on SMP parallel computers. First published in 1997 for the Fortran language and in 1998 for the C and C++ languages, the latest version (OpenMP Version 3.0) was released in May 2008. OpenMP does not support Java. The official standard and a wealth of information about OpenMP are available on the OpenMP.org Web site. Numerous free implementations and commercial implementations of the OpenMP standard are available.

To write an OpenMP program, you take a normal Fortran, C, or C++ program and add **pragmas** designating where and how the program is to be executed in parallel. There are also OpenMP subroutines you can call for various purposes. Then, you run your program through a special OpenMP compiler. The OpenMP compiler looks at the pragmas; it rewrites your program to add threads, shared variables, per-thread variables, barriers, and so on as directed by the pragmas; and it compiles the now-multithreaded program. When the compiled program runs on an SMP parallel computer, the program runs in multiple threads, resulting in a parallel speedup (one hopes).

As a simple example, here is the key portion of a sequential C program for Floyd's all-shortest-paths algorithm.

```
int n; // Number of vertices
double **d; // Distance matrix
int i, r, c;

for (i = 0; i < n; ++ i)
  {
  double *d_i = d[i];
  for (r = 0; r < n; ++ r)
    {
    double *d_r = d[r];
    for (c = 0; c < n; ++ c)
      {
      d_r[c] = min (d_r[c], d_r[i] + d_i[c]);
      }
    }
 }
```

And here is the OpenMP parallel version.

```
int n; // Number of vertices
double **d; // Distance matrix

#pragma omp parallel
   {
   int i, r, c;
   for (i = 0; i < n; ++ i)
      {
      double *d_i = d[i];
      #pragma omp for
      for (r = 0; r < n; ++ r)
         {
         double *d_r = d[r];
         for (c = 0; c < n; ++ c)
            {
            d_r[c] = min (d_r[c], d_r[i] + d_i[c]);
            }
         }
      }
   }
```

For the complete C source files, see the Parallel Java Library documentation (Javadoc). The source files are linked from the package summary page for package edu.rit.smp.network.

The `#pragma omp` signals an OpenMP pragma. The `parallel` pragma states that the following block of code is to be executed in parallel by a team of threads. Because the number of threads is not specified in the pragma, the number of threads is determined at run time. Inside the parallel region, each thread gets its own copies of the `i`, `r`, and `c` variables; that is, these are per-thread variables. The other variables `n` and `d`, which are declared outside the parallel region, are shared variables.

All the threads execute the outer loop. However, when they reach the middle loop, the `for` pragma states that the middle loop is to be executed as a work-sharing parallel loop. That is, the middle loop iterations are partitioned, and each thread executes a different subset of the iterations. Because no loop schedule is specified, a default schedule is used. At the end of the middle loop, all the threads do an implicit barrier wait before proceeding to the next outer loop iteration.

## A.2 OpenMP Features

Parallel Java's SMP parallel programming features are inspired by OpenMP. A comparison of the principal features follows. A complete list of OpenMP features and Parallel Java features is too lengthy to include here. For further information, refer to the OpenMP standard and the Parallel Java documentation.

**Region of parallel code**. OpenMP uses the `parallel` pragma.

```
#pragma omp parallel
   {
   // Parallel code
   }
```

Parallel Java uses instances of class ParallelTeam and ParallelRegion.

```
new ParallelTeam().execute (new ParallelRegion()
   {
   public void run()
      {
      // Parallel code
      }
   });
```

**Number of threads**. OpenMP specifies the number of threads in a parallel team either at compile time in the `parallel` pragma or, if not specified, at run time with the `OMP_NUM_THREADS` environment variable.

```
#pragma omp parallel num_threads(2)
```

Parallel Java specifies the number of threads in a parallel team either at compile time as the ParallelTeam constructor argument or, if not specified, at run time with the `-Dpj.nt` property.

```
new ParallelTeam(2).execute ...
```

**Nested parallel regions**. OpenMP allows parallel regions to be nested inside each other with certain restrictions; typically, there is a limit on the number of levels of nesting allowed. Parallel Java allows parallel teams and parallel regions to be nested inside each other with no restrictions.

**Shared variables**. In OpenMP, variables declared outside a parallel region are shared among the parallel team threads. Variables declared inside a parallel region are normally per-thread variables, but may instead be shared by listing them in the `parallel` pragma. For example, here `x` is a shared variable.

```
#pragma omp parallel shared(x)
   {
   int x;
   x = ...
   }
```

In Parallel Java, a variable must be declared outside a parallel region to be a shared variable. Neither OpenMP nor Parallel Java automatically synchronize accesses to a shared variable; it is up to the programmer to include the proper synchronization when accessing the variable (or to decide that synchronization is not needed).

**Per-thread variables**. In OpenMP, variables declared inside a parallel region—"private" variables in OpenMP parlance—are not shared among the parallel team threads; instead, each thread gets its own copy of the variable. Variables declared outside a parallel region are normally shared variables, but may instead be private by listing them in the `parallel` pragma. For example, here `x` is a private variable.

```
int x;
#pragma omp parallel private(x)
   {
   x = ...
   }
```

In Parallel Java, a variable must be declared inside a parallel region to be a per-thread variable. (They're not called private variables because "private" has a different connotation in Java.)

**Work-sharing parallel loops**. OpenMP uses the `for` pragma.

```
#pragma omp parallel
   {
   int i;
   #pragma omp for
   for (i = 0; i <= 99; ++ i)
      {
      // Loop body
      }
   }
```

Parallel Java uses an instance of class IntegerForLoop or LongForLoop.

```
new ParallelTeam().execute (new ParallelRegion()
   {
   public void run()
      {
      execute (0, 99, new IntegerForLoop()
         {
         public void run (int first, int last)
            {
            for (int i = first; i <= last; ++ i)
               {
               // Loop body
               }
            }
         });
      }
   });
```

Both OpenMP and Parallel Java allow loop indexes of type `int` and type `long` and allow strides greater than 1. OpenMP also allows the stride to be negative (that is, the loop index counts down).

**Parallel loop schedules**. OpenMP specifies a parallel loop's schedule either at compile time in the `for` pragma or, if not specified, at run time with the `OMP_SCHEDULE` environment variable.

```
#pragma omp for schedule(guided)
for (i = 0; i <= 99; ++ i)
   {
   // Loop body
   }
```

Parallel Java specifies a parallel loop's schedule either at compile time by defining the `schedule()` method or, if not specified, at run time with the `-Dpj.schedule` property.

```
execute (0, 99, new IntegerForLoop()
   {
   public IntegerSchedule schedule()
      {
      return IntegerSchedule.guided();
      }
   });
```

Both OpenMP and Parallel Java support fixed, dynamic, guided, and runtime schedules. (What Parallel Java calls a "fixed" schedule, OpenMP calls a "static" schedule. "Static" has a different connotation in Java.) Parallel Java also supports arbitrary user-defined parallel loop schedules; simply define a subclass of class IntegerSchedule or LongSchedule with the desired partitioning algorithm.

**Parallel iterations**. Parallel Java has the ability to iterate in parallel over the elements of an array, the items in an Iterable collection, or the items returned by an Iterator. This is the parallel analog of the Java language's for-each loop construct. (We have not used parallel iterations in the programs in this book.) Because the Fortran, C, and C++ languages lack the for-each loop construct, OpenMP does not support parallel iterations.

**Parallel section groups**. To get different threads to execute different sections of code—such as a computation section and a file output section—OpenMP uses the `sections` and `section` pragmas.

```
#pragma omp parallel num_threads(2)
   {
   #pragma omp sections
      {
      #pragma omp section
         {
         // Computation code
         }
      #pragma omp section
         {
         // File output code
         }
      }
   }
```

Parallel Java executes one or more instances of class ParallelSection.

```
new ParallelTeam(2).execute (new ParallelRegion()
   {
   public void run()
      {
      execute (new ParallelSection()
         {
         public void run()
            {
            // Computation code
            }
         },
      new ParallelSection()
         {
         public void run()
            {
            // File output code
            }
         });
      }
   });
```

**Reduction variables**. To support the *reduction* pattern, OpenMP lets a shared variable be designated as a reduction variable in the `parallel` pragma. Each thread gets its own copy of the variable. Inside the parallel region, the variable's name refers to the per-thread copy, so each thread updates its own copy without conflicting with the other threads. At the end of the parallel region, the per-thread variables are automatically reduced together into the shared variable using a reduction operator specified in the `parallel` pragma. Outside the parallel region, the variable's name refers to the shared variable. For example, here `count` is a reduction variable, and addition (+) is the reduction operator.

```
long long int count = 0;
#pragma omp parallel reduction(+:count)
   {
   int i;
   for (i = 0; i < 1000000; ++ i)
      {
      if (...) ++ count;
      }
   }
printf ("%lld\n", count);
```

In Parallel Java, the shared reduction variable and the per-thread variables must be declared separately, the shared variable must be multiple thread safe, and the reduction operation must be coded explicitly.

```
SharedLong count = new SharedLong (0);
new ParallelTeam().execute (new ParallelRegion()
   {
   public void run()
      {
      long thrCount = 0;
      for (int i = 0; i < 1000000; ++ i)
         {
         if (...) ++ thrCount;
         }
      count.addAndGet (thrCount);
      }
   });
System.out.println (count);
```

**Reduction operators**. OpenMP supports only certain reduction operators (`+ * - & | ^ && ||` in C and C++) and supports reduction only on primitive types. C++ overloaded operators are not supported.

Parallel Java provides several predefined reduction operators as well as supporting arbitrary user-defined reduction operators; just define an appropriate subclass of class edu.rit.pj.reduction.Op. Parallel Java supports reduction on primitive types, on arrays of primitive types, on arbitrary object types, and on arrays of object types.

## A.3 OpenMP Performance

Some folks think that Java is not a suitable language for high-performance computing (HPC) because Java programs are too slow in comparison to the established HPC languages, Fortran, C, and, to some extent, C++. This was true last century, when Java was mainly an interpreted language. But modern JVMs use sophisticated just-in-time (JIT) compilers to convert Java bytecode to highly optimized machine code, which is then executed. Consequently, Java programs today can run as fast as or even faster than, say, C programs.

To get a sense of how a parallel C program's performance compares with a parallel Java program's performance, sequential and parallel versions of Floyd's Algorithm were implemented in C with OpenMP and in Java with Parallel Java. To make the comparison fair to Java, the C program included statements to do array index bounds checking, which Java does automatically, but C does not. To make the comparison fair to C, the C program was compiled with the highest level of optimization, which the JVM's JIT compiler does automatically.

The C and Java programs were compiled and run on the "parasite" machine, a Sun Microsystems eight-processor SMP parallel computer with four UltraSPARC-IV dual-core CPU chips, a 1.35 GHz CPU clock speed, and 16 GB of main memory. The C programs were compiled with the Sun C compiler at optimization level 5 (the highest possible). The Java programs were run with the Sun JDK 1.5.0_15 HotSpot Server Virtual Machine. Table A.1 (at the end of the appendix) lists, and Figure A.1 plots, the C program's and the Java program's performance. The running times are for the calculation portion only.

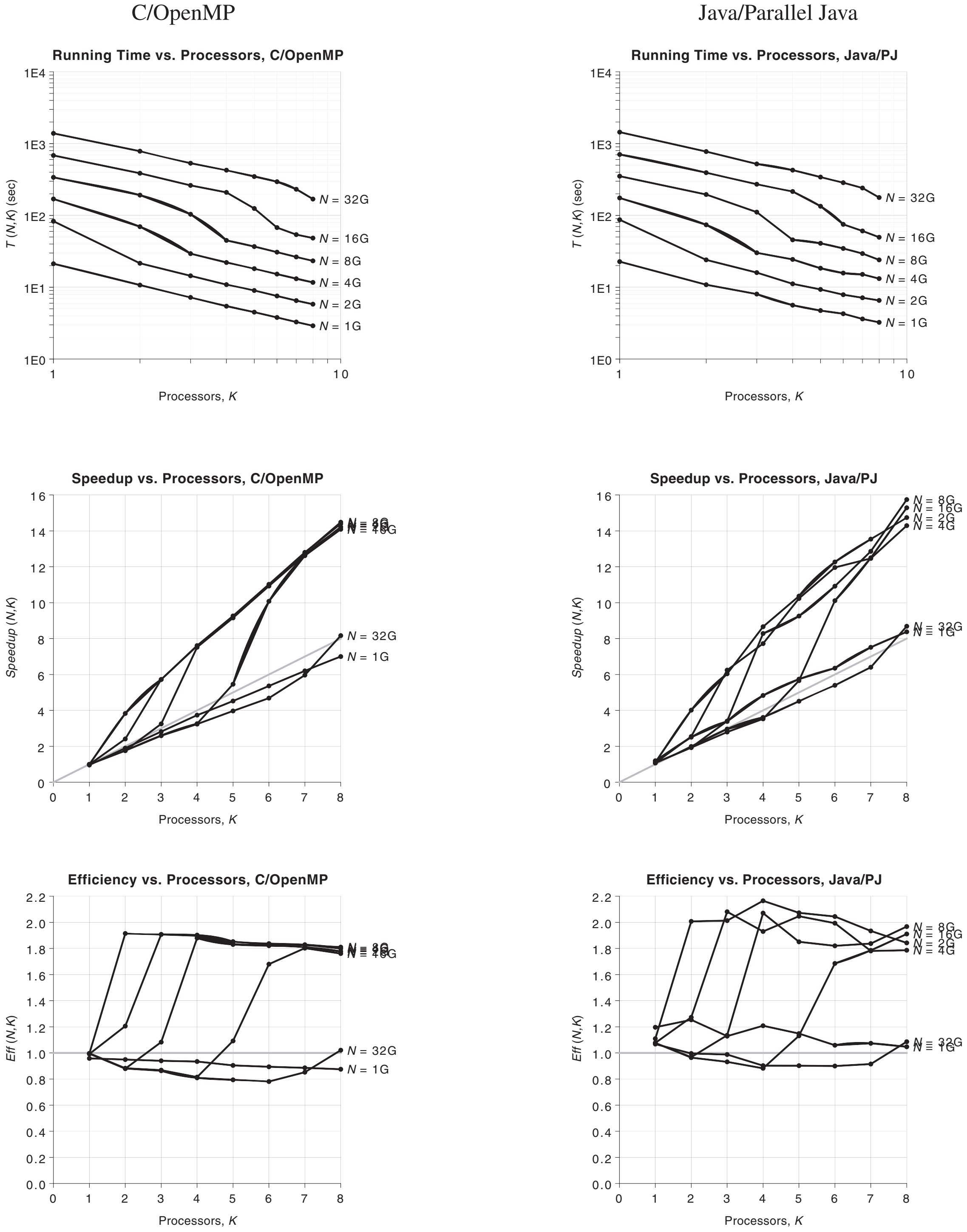


**Figure A.1** Floyd's Algorithm program running-time metrics

The program was run on distance matrices with the numbers of vertices $n$ and problem sizes $N = n^3$ as follows:

| $n$ | $N$ | |
|---:|---:|---:|
| 1,000 | 1,000,000,000 | (1G) |
| 1,260 | 2,000,376,000 | (2G) |
| 1,590 | 4,019,679,000 | (4G) |
| 2,000 | 8,000,000,000 | (8G) |
| 2,520 | 16,003,008,000 | (16G) |
| 3,180 | 32,157,432,000 | (32G) |

The Java program's running times range from 2 percent below to 15 percent above the C program's running times. Both the C and the Java versions experience efficiencies around 2 due to cache effects, as explained in Chapter 16.

As a further comparison between C and Java, sequential and SMP parallel versions of the Monte Carlo program for estimating π (see Chapter 14) were implemented in C with OpenMP and in Java with Parallel Java. Where the Floyd's Algorithm program has a lot of thread synchronization (a barrier wait at the end of every outer loop iteration), the π estimating program has little thread synchronization (one reduction at the end of the program). For the complete C source files, see the Parallel Java Library documentation (Javadoc). The source files are linked from the package summary page for package edu.rit.smp.monte.

The C and Java programs were compiled and run on the "parasite" machine with $N$ = 1, 2, 5, 10, 20, and 50 billion darts. Table A.2 (at the end of the appendix) lists, and Figure A.2 plots, the C program's and the Java program's performance. The Java program's running times are 40 percent smaller than the C program's running times.

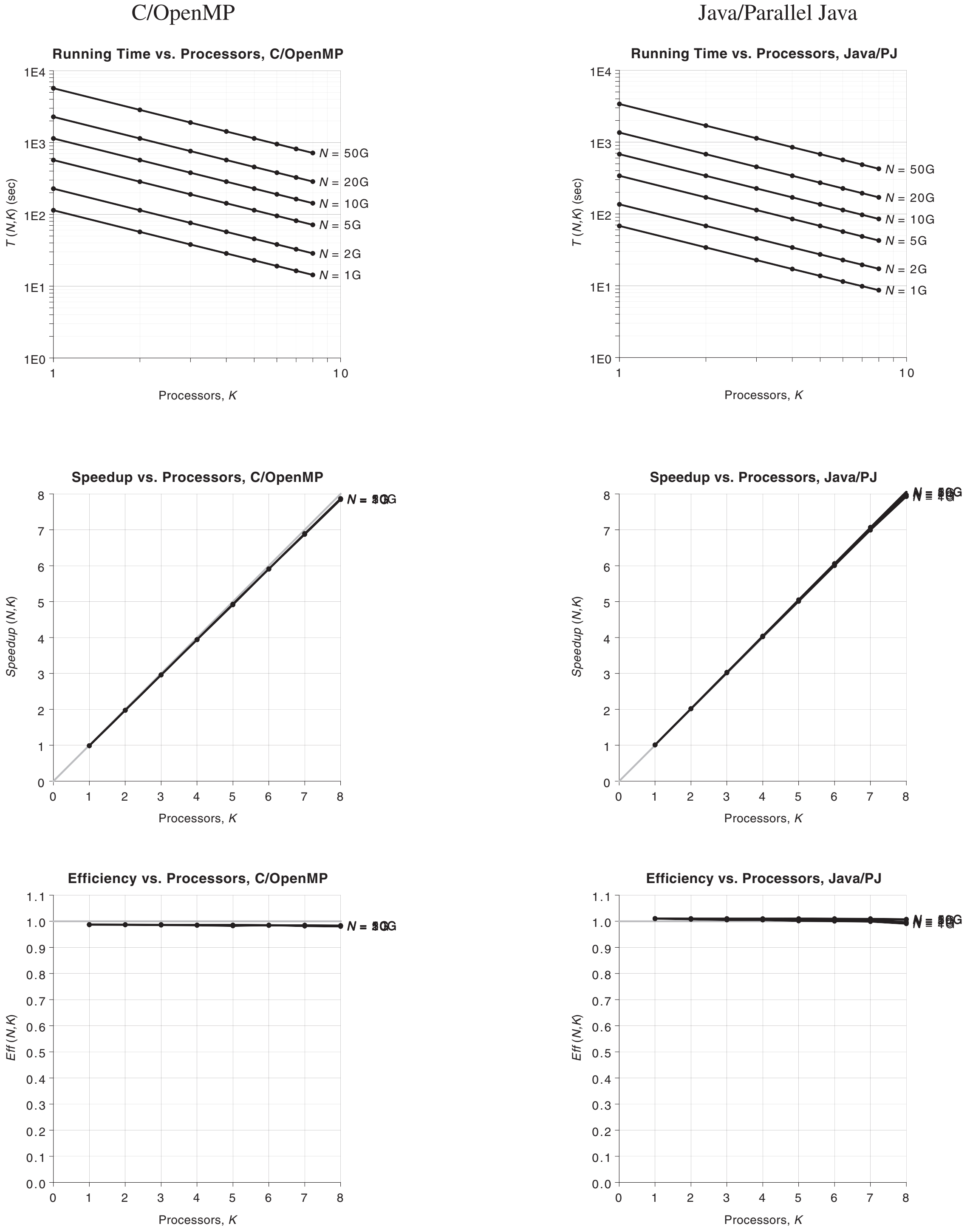


**Figure A.2** π estimating program running-time metrics

## A.4 For Further Information

On the official OpenMP standard:

- OpenMP.org Web Site. http://openmp.org/wp/
- OpenMP Architecture Review Board. *OpenMP Application Program Interface, Version 3.0*. May 2008. http://www.openmp.org/mp-documents/spec30.pdf

Textbooks on parallel programming with OpenMP:

- B. Chapman, G. Yost, and R. van der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming*. MIT Press, 2008.
- M. Quinn. *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill, 2004.
- R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, and R. Menon. *Parallel Programming in OpenMP*. Academic Press, 2001.

**Table A.1** Floyd's Algorithm program running-time metrics

| C/OpenMP | | | | | | Java/Parallel Java | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| *N* | *K* | *T* | *Spdup* | *Eff* | *EDSF* | *N* | *K* | *T* | *Spdup* | *Eff* | *EDSF* |
| 1G | seq | 20349 | | | | 1G | seq | 27191 | | | |
| 1G | 1 | 21238 | 0.958 | 0.958 | | 1G | 1 | 22726 | 1.196 | 1.196 | |
| 1G | 2 | 10718 | 1.899 | 0.949 | 0.009 | 1G | 2 | 10847 | 2.507 | 1.253 | -0.045 |
| 1G | 3 | 7218 | 2.819 | 0.940 | 0.010 | 1G | 3 | 8032 | 3.385 | 1.128 | 0.030 |
| 1G | 4 | 5446 | 3.737 | 0.934 | 0.009 | 1G | 4 | 5627 | 4.832 | 1.208 | -0.003 |
| 1G | 5 | 4500 | 4.522 | 0.904 | 0.015 | 1G | 5 | 4736 | 5.741 | 1.148 | 0.010 |
| 1G | 6 | 3794 | 5.363 | 0.894 | 0.014 | 1G | 6 | 4280 | 6.353 | 1.059 | 0.026 |
| 1G | 7 | 3282 | 6.200 | 0.886 | 0.014 | 1G | 7 | 3618 | 7.515 | 1.074 | 0.019 |
| 1G | 8 | 2907 | 7.000 | 0.875 | 0.014 | 1G | 8 | 3246 | 8.377 | 1.047 | 0.020 |
| 2G | seq | 82572 | | | | 2G | seq | 96580 | | | |
| 2G | 1 | 83007 | 0.995 | 0.995 | | 2G | 1 | 87071 | 1.109 | 1.109 | |
| 2G | 2 | 21563 | 3.829 | 1.915 | -0.480 | 2G | 2 | 24056 | 4.015 | 2.007 | -0.447 |
| 2G | 3 | 14430 | 5.722 | 1.907 | -0.239 | 2G | 3 | 15991 | 6.040 | 2.013 | -0.225 |
| 2G | 4 | 10879 | 7.590 | 1.898 | -0.159 | 2G | 4 | 11152 | 8.660 | 2.165 | -0.163 |
| 2G | 5 | 9008 | 9.167 | 1.833 | -0.114 | 2G | 5 | 9318 | 10.365 | 2.073 | -0.116 |
| 2G | 6 | 7557 | 10.927 | 1.821 | -0.091 | 2G | 6 | 7873 | 12.267 | 2.045 | -0.091 |
| 2G | 7 | 6520 | 12.664 | 1.809 | -0.075 | 2G | 7 | 7133 | 13.540 | 1.934 | -0.071 |
| 2G | 8 | 5808 | 14.217 | 1.777 | -0.063 | 2G | 8 | 6552 | 14.741 | 1.843 | -0.057 |

**Table A.1** Floyd's Algorithm program running-time metrics (cont.)

| C/OpenMP | | | | | | Java/Parallel Java | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| *N* | *K* | *T* | *Spdup* | *Eff* | *EDSF* | *N* | *K* | *T* | *Spdup* | *Eff* | *EDSF* |
| 4G | seq | 168073 | | | | 4G | seq | 188427 | | | |
| 4G | 1 | 169451 | 0.992 | 0.992 | | 4G | 1 | 175013 | 1.077 | 1.077 | |
| 4G | 2 | 69721 | 2.411 | 1.205 | -0.177 | 4G | 2 | 74020 | 2.546 | 1.273 | -0.154 |
| 4G | 3 | 29371 | 5.722 | 1.907 | -0.240 | 4G | 3 | 30180 | 6.243 | 2.081 | -0.241 |
| 4G | 4 | 22086 | 7.610 | 1.902 | -0.160 | 4G | 4 | 24405 | 7.721 | 1.930 | -0.147 |
| 4G | 5 | 18147 | 9.262 | 1.852 | -0.116 | 4G | 5 | 18411 | 10.234 | 2.047 | -0.119 |
| 4G | 6 | 15248 | 11.023 | 1.837 | -0.092 | 4G | 6 | 15753 | 11.961 | 1.994 | -0.092 |
| 4G | 7 | 13129 | 12.802 | 1.829 | -0.076 | 4G | 7 | 15115 | 12.466 | 1.781 | -0.066 |
| 4G | 8 | 11676 | 14.395 | 1.799 | -0.064 | 4G | 8 | 13185 | 14.291 | 1.786 | -0.057 |
| 8G | seq | 337396 | | | | 8G | seq | 378557 | | | |
| 8G | 1 | 339047 | 0.995 | 0.995 | | 8G | 1 | 351896 | 1.076 | 1.076 | |
| 8G | 2 | 191853 | 1.759 | 0.879 | 0.132 | 8G | 2 | 195594 | 1.935 | 0.968 | 0.112 |
| 8G | 3 | 103852 | 3.249 | 1.083 | -0.041 | 8G | 3 | 111168 | 3.405 | 1.135 | -0.026 |
| 8G | 4 | 44878 | 7.518 | 1.880 | -0.157 | 8G | 4 | 45701 | 8.283 | 2.071 | -0.160 |
| 8G | 5 | 36861 | 9.153 | 1.831 | -0.114 | 8G | 5 | 40903 | 9.255 | 1.851 | -0.105 |
| 8G | 6 | 30805 | 10.953 | 1.825 | -0.091 | 8G | 6 | 34665 | 10.920 | 1.820 | -0.082 |
| 8G | 7 | 26410 | 12.775 | 1.825 | -0.076 | 8G | 7 | 29432 | 12.862 | 1.837 | -0.069 |
| 8G | 8 | 23297 | 14.482 | 1.810 | -0.064 | 8G | 8 | 24048 | 15.742 | 1.968 | -0.065 |
| 16G | seq | 682138 | | | | 16G | seq | 759572 | | | |
| 16G | 1 | 684118 | 0.997 | 0.997 | | 16G | 1 | 705772 | 1.076 | 1.076 | |
| 16G | 2 | 386430 | 1.765 | 0.883 | 0.130 | 16G | 2 | 393951 | 1.928 | 0.964 | 0.116 |
| 16G | 3 | 261972 | 2.604 | 0.868 | 0.074 | 16G | 3 | 271809 | 2.795 | 0.932 | 0.078 |
| 16G | 4 | 209303 | 3.259 | 0.815 | 0.075 | 16G | 4 | 215268 | 3.528 | 0.882 | 0.073 |
| 16G | 5 | 124963 | 5.459 | 1.092 | -0.022 | 16G | 5 | 134310 | 5.655 | 1.131 | -0.012 |
| 16G | 6 | 67686 | 10.078 | 1.680 | -0.081 | 16G | 6 | 75120 | 10.111 | 1.685 | -0.072 |
| 16G | 7 | 54066 | 12.617 | 1.802 | -0.074 | 16G | 7 | 60768 | 12.500 | 1.786 | -0.066 |
| 16G | 8 | 48418 | 14.089 | 1.761 | -0.062 | 16G | 8 | 49690 | 15.286 | 1.911 | -0.062 |
| 32G | seq | 1380749 | | | | 32G | seq | 1542463 | | | |
| 32G | 1 | 1388991 | 0.994 | 0.994 | | 32G | 1 | 1444761 | 1.068 | 1.068 | |
| 32G | 2 | 783923 | 1.761 | 0.881 | 0.129 | 32G | 2 | 774616 | 1.991 | 0.996 | 0.072 |
| 32G | 3 | 533088 | 2.590 | 0.863 | 0.076 | 32G | 3 | 520599 | 2.963 | 0.988 | 0.041 |
| 32G | 4 | 426584 | 3.237 | 0.809 | 0.076 | 32G | 4 | 427279 | 3.610 | 0.902 | 0.061 |
| 32G | 5 | 347860 | 3.969 | 0.794 | 0.063 | 32G | 5 | 341923 | 4.511 | 0.902 | 0.046 |
| 32G | 6 | 294745 | 4.685 | 0.781 | 0.055 | 32G | 6 | 285792 | 5.397 | 0.900 | 0.037 |
| 32G | 7 | 231481 | 5.965 | 0.852 | 0.028 | 32G | 7 | 240887 | 6.403 | 0.915 | 0.028 |
| 32G | 8 | 169085 | 8.166 | 1.021 | -0.004 | 32G | 8 | 177570 | 8.687 | 1.086 | -0.002 |

**Table A.2** $\pi$ estimating program running-time metrics

| C/OpenMP | | | | | | Java/Parallel Java | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| *N* | *K* | *T* | *Spdup* | *Eff* | *EDSF* | *N* | *K* | *T* | *Spdup* | *Eff* | *EDSF* |
| 1G | seq | 112473 | | | | 1G | seq | 68702 | | | |
| 1G | 1 | 113991 | 0.987 | 0.987 | | 1G | 1 | 68030 | 1.010 | 1.010 | |
| 1G | 2 | 57011 | 1.973 | 0.986 | 0.000 | 1G | 2 | 34099 | 2.015 | 1.007 | 0.002 |
| 1G | 3 | 38054 | 2.956 | 0.985 | 0.001 | 1G | 3 | 22789 | 3.015 | 1.005 | 0.002 |
| 1G | 4 | 28565 | 3.937 | 0.984 | 0.001 | 1G | 4 | 17094 | 4.019 | 1.005 | 0.002 |
| 1G | 5 | 22900 | 4.911 | 0.982 | 0.001 | 1G | 5 | 13722 | 5.007 | 1.001 | 0.002 |
| 1G | 6 | 19031 | 5.910 | 0.985 | 0.000 | 1G | 6 | 11447 | 6.002 | 1.000 | 0.002 |
| 1G | 7 | 16369 | 6.871 | 0.982 | 0.001 | 1G | 7 | 9827 | 6.991 | 0.999 | 0.002 |
| 1G | 8 | 14344 | 7.841 | 0.980 | 0.001 | 1G | 8 | 8669 | 7.925 | 0.991 | 0.003 |
| 2G | seq | 225036 | | | | 2G | seq | 137300 | | | |
| 2G | 1 | 227885 | 0.987 | 0.987 | | 2G | 1 | 135906 | 1.010 | 1.010 | |
| 2G | 2 | 114021 | 1.974 | 0.987 | 0.001 | 2G | 2 | 68046 | 2.018 | 1.009 | 0.001 |
| 2G | 3 | 76030 | 2.960 | 0.987 | 0.000 | 2G | 3 | 45408 | 3.024 | 1.008 | 0.001 |
| 2G | 4 | 57133 | 3.939 | 0.985 | 0.001 | 2G | 4 | 34106 | 4.026 | 1.006 | 0.001 |
| 2G | 5 | 45681 | 4.926 | 0.985 | 0.001 | 2G | 5 | 27303 | 5.029 | 1.006 | 0.001 |
| 2G | 6 | 38123 | 5.903 | 0.984 | 0.001 | 2G | 6 | 22785 | 6.026 | 1.004 | 0.001 |
| 2G | 7 | 32669 | 6.888 | 0.984 | 0.001 | 2G | 7 | 19543 | 7.026 | 1.004 | 0.001 |
| 2G | 8 | 28624 | 7.862 | 0.983 | 0.001 | 2G | 8 | 17220 | 7.973 | 0.997 | 0.002 |
| 5G | seq | 562548 | | | | 5G | seq | 343064 | | | |
| 5G | 1 | 569801 | 0.987 | 0.987 | | 5G | 1 | 339525 | 1.010 | 1.010 | |
| 5G | 2 | 284986 | 1.974 | 0.987 | 0.000 | 5G | 2 | 169879 | 2.019 | 1.010 | 0.001 |
| 5G | 3 | 189962 | 2.961 | 0.987 | 0.000 | 5G | 3 | 113296 | 3.028 | 1.009 | 0.001 |
| 5G | 4 | 142706 | 3.942 | 0.986 | 0.001 | 5G | 4 | 85035 | 4.034 | 1.009 | 0.001 |
| 5G | 5 | 114207 | 4.926 | 0.985 | 0.001 | 5G | 5 | 68046 | 5.042 | 1.008 | 0.001 |
| 5G | 6 | 95221 | 5.908 | 0.985 | 0.001 | 5G | 6 | 56727 | 6.048 | 1.008 | 0.000 |
| 5G | 7 | 81583 | 6.895 | 0.985 | 0.000 | 5G | 7 | 48651 | 7.052 | 1.007 | 0.001 |
| 5G | 8 | 71488 | 7.869 | 0.984 | 0.001 | 5G | 8 | 42639 | 8.046 | 1.006 | 0.001 |
| 10G | seq | 1125138 | | | | 10G | seq | 685992 | | | |
| 10G | 1 | 1139809 | 0.987 | 0.987 | | 10G | 1 | 678884 | 1.010 | 1.010 | |
| 10G | 2 | 569906 | 1.974 | 0.987 | 0.000 | 10G | 2 | 339580 | 2.020 | 1.010 | 0.000 |
| 10G | 3 | 379930 | 2.961 | 0.987 | 0.000 | 10G | 3 | 226441 | 3.029 | 1.010 | 0.000 |
| 10G | 4 | 285463 | 3.941 | 0.985 | 0.001 | 10G | 4 | 169925 | 4.037 | 1.009 | 0.000 |
| 10G | 5 | 228383 | 4.927 | 0.985 | 0.000 | 10G | 5 | 135964 | 5.045 | 1.009 | 0.000 |
| 10G | 6 | 190315 | 5.912 | 0.985 | 0.000 | 10G | 6 | 113319 | 6.054 | 1.009 | 0.000 |
| 10G | 7 | 163217 | 6.894 | 0.985 | 0.000 | 10G | 7 | 97132 | 7.062 | 1.009 | 0.000 |
| 10G | 8 | 142959 | 7.870 | 0.984 | 0.000 | 10G | 8 | 85158 | 8.056 | 1.007 | 0.001 |

**Table A.2** π estimating program running-time metrics (cont.)

| C/OpenMP | | | | | | Java/Parallel Java | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| *N* | *K* | *T* | *Spdup* | *Eff* | *EDSF* | *N* | *K* | *T* | *Spdup* | *Eff* | *EDSF* |
| 20G | seq | 2250239 | | | | 20G | seq | 1371888 | | | |
| 20G | 1 | 2279651 | 0.987 | 0.987 | | 20G | 1 | 1357531 | 1.011 | 1.011 | |
| 20G | 2 | 1139833 | 1.974 | 0.987 | 0.000 | 20G | 2 | 678981 | 2.021 | 1.010 | 0.000 |
| 20G | 3 | 759883 | 2.961 | 0.987 | 0.000 | 20G | 3 | 452664 | 3.031 | 1.010 | 0.000 |
| 20G | 4 | 570809 | 3.942 | 0.986 | 0.001 | 20G | 4 | 339668 | 4.039 | 1.010 | 0.000 |
| 20G | 5 | 456722 | 4.927 | 0.985 | 0.000 | 20G | 5 | 271701 | 5.049 | 1.010 | 0.000 |
| 20G | 6 | 380587 | 5.913 | 0.985 | 0.000 | 20G | 6 | 226480 | 6.057 | 1.010 | 0.000 |
| 20G | 7 | 326392 | 6.894 | 0.985 | 0.000 | 20G | 7 | 194156 | 7.066 | 1.009 | 0.000 |
| 20G | 8 | 286297 | 7.860 | 0.982 | 0.001 | 20G | 8 | 170231 | 8.059 | 1.007 | 0.000 |
| 50G | seq | 5625605 | | | | 50G | seq | 3429530 | | | |
| 50G | 1 | 5699208 | 0.987 | 0.987 | | 50G | 1 | 3393498 | 1.011 | 1.011 | |
| 50G | 2 | 2849585 | 1.974 | 0.987 | 0.000 | 50G | 2 | 1696993 | 2.021 | 1.010 | 0.000 |
| 50G | 3 | 1899721 | 2.961 | 0.987 | 0.000 | 50G | 3 | 1131360 | 3.031 | 1.010 | 0.000 |
| 50G | 4 | 1427108 | 3.942 | 0.985 | 0.001 | 50G | 4 | 848870 | 4.040 | 1.010 | 0.000 |
| 50G | 5 | 1141660 | 4.928 | 0.986 | 0.000 | 50G | 5 | 679023 | 5.051 | 1.010 | 0.000 |
| 50G | 6 | 951528 | 5.912 | 0.985 | 0.000 | 50G | 6 | 565910 | 6.060 | 1.010 | 0.000 |
| 50G | 7 | 815872 | 6.895 | 0.985 | 0.000 | 50G | 7 | 485098 | 7.070 | 1.010 | 0.000 |
| 50G | 8 | 715684 | 7.860 | 0.983 | 0.001 | 50G | 8 | 425339 | 8.063 | 1.008 | 0.000 |

# APPENDIX B

# Message Passing Interface (MPI)

in which we take a quick look at cluster parallel programming with MPI; and we compare MPI’s features and performance to Parallel Java

# APPENDIX B Message Passing Interface (MPI)

## B.1 MPI Programming

Message Passing Interface (MPI) is a standard application program interface (API) for writing parallel programs consisting of multiple processes that communicate by sending and receiving messages. An MPI program can run on a cluster parallel computer, with one process on each node and messages traversing the network between nodes. An MPI program can also run on an SMP parallel computer, with one process on each CPU and messages going between processes through shared memory in the operating system.

MPI Version 1.0, supporting the Fortran 77 and C programming languages, was released in 1994. MPI Version 1.1 was released a year later. The next major version, MPI Version 2.0, supporting Fortran 77, Fortran 90, C, and C++, was released in 1997. At the time of this writing, MPI Version 2.0 is the official standard. Work on another revision is underway; a draft of MPI Version 2.1 was released in June 2008. MPI does not support Java. The official MPI standards are available on the Message Passing Interface Forum Web site. Numerous free implementations and commercial implementations of the MPI standard are available.

Unlike OpenMP which uses pragmas and a special compiler, MPI is just a subroutine library—a *large* subroutine library. To write an MPI program, you write a normal Fortran, C, or C++ program and include calls to the MPI subroutines to send and receive messages. Then, you run your program through the regular Fortran, C, or C++ compiler. When the compiled program is executed on a parallel computer, typically via a special MPI launcher application, the program runs in multiple processes on the nodes of the parallel computer, resulting in a speedup (one hopes).

As a simple example, here are a few key portions of class FloydClu, a cluster parallel program for Floyd's all-shortest-paths algorithm in Java with Parallel Java (see Chapter 25 for the complete program).

```
static Comm world; // World communicator
static int size;
static int rank;

static int n; // Number of nodes
static double[][] d; // Distance matrix

static double[] row_i; // Row broadcast from another process
static DoubleBuf row_i_buf;

   Comm.init (args);
   world = Comm.world();
   size = world.size();
   rank = world.rank();

   for (int i = 0; i < n; ++ i)
      {
      double[] d_i = d[i];
      if (rank == i_root)
         {
         world.broadcast (i_root, DoubleBuf.buffer (d_i));
         }
      else
         {
         world.broadcast (i_root, row_i_buf);
         d_i = row_i;
         }
      for (int r = mylb; r <= myub; ++ r)
         {
         double[] d_r = d[r];
         for (int c = 0; c < n; ++ c)
            {
            d_r[c] = Math.min (d_r[c], d_r[i] + d_i[c]);
            }
         }
      }
```

And here are the equivalent portions of a cluster parallel program written in C with MPI.

```
static MPI_Comm world; // World communicator
static int size;
static int rank;

static int n; // Number of nodes
static double **d; // Distance matrix

static double *row_i; // Row broadcast from another process

   MPI_Init (&argc, &argv);
   world = MPI_COMM_WORLD;
   MPI_Comm_size (world, &size);
   MPI_Comm_rank (world, &rank);

   for (int i = 0; i < n; ++ i)
      {
      double *d_i = d[i];
      if (rank == i_root)
```

```
         {
         MPI_Bcast (d_i, n, MPI_DOUBLE, i_root, world);
         }
      else
         {
         MPI_Bcast (row_i, n, MPI_DOUBLE, i_root, world);
         d_i = row_i;
         }
      for (int r = mylb; r <= myub; ++ r)
         {
         double[] d_r = d[r];
         for (int c = 0; c < n; ++ c)
            {
            d_r[c] = Math.min (d_r[c], d_r[i] + d_i[c]);
            }
         }
      }

   MPI_Finalize();
```

For the complete C source files, see the Parallel Java Library documentation (Javadoc). The source files are linked from the package summary page for package edu.rit.clu.network.

The variable `world` of type `MPI_Comm` is the world communicator, which encompasses all the processes in the program. The program calls the MPI subroutines `MPI_Init()` to initialize the world communicator, `MPI_Size()` to find the number of processes in the program *K*, and `MPI_Rank()` to find the current process's rank in the range 0 through *K*–1. The `MPI_Bcast()` subroutine broadcasts a message to all processes. The arguments of `MPI_Bcast()` are as follows:

- The address of the first data item to include in the message, either `d_i` (the *i*-th row of the distance matrix) if this process is sending, or `row_i` (an extra row's worth of storage) if this process is receiving.
- The number of data items to include in the message, `n`.
- The type of the data items, type `double` (designated by the constant `MPI_DOUBLE`).
- The rank of the root process for the broadcast (the source process that has the data items to be sent), `i_root`.
- The communicator in which to perform the broadcast, the world communicator.

As its last act, the program must call the `MPI_Finalize()` subroutine; otherwise, a run-time error will occur.

## B.2 MPI Features

Parallel Java's communicator object, class edu.rit.pj.Comm, and its message passing methods are inspired by MPI. Indeed, every Parallel Java message passing operation is found in MPI, including the following:

- Point-to-point communication operations: send; receive; receive from a wildcard source; send-receive; message tags; and nonblocking versions of send, receive, and send-receive.
- Collective communication operations: broadcast, scatter, gather, all-gather, reduce, all-reduce, all-to-all, scan, exclusive scan, and barrier.
- Reduction with predefined reduction operators and user-defined reduction operators.
- The ability to create a new communicator encompassing a subset of the processes in an existing communicator.

MPI has many other message passing subroutines—too many to include here. For further information, refer to the MPI standard and the Parallel Java documentation.

Although Parallel Java's message passing *functionality* is the same as MPI's, Parallel Java's *API design* differs from MPI's. Parallel Java's API is organized around two aspects: message passing operations; and data sources and destinations. Message passing operations are expressed as methods of class Comm. Data sources and destinations are expressed as buffer objects, instances of subclasses of class edu.rit.mp.Buf. The two aspects are orthogonal; any buffer object can be used in any message passing operation.

Keeping these two aspects separate simplifies the API. For example, in Parallel Java, there is one scatter method with three arguments.

```
scatter (int root, Buf[] srcarray, Buf dst);
```

But in MPI, there are two scatter subroutines.

```
MPI_Scatter
   (void *sendbuf, int sendcount, MPI_Datatype sendtype,
    void *recvbuf, int recvcount, MPI_Datatype recvtype,
    int root, MPI_Comm comm);
MPI_Scatterv
   (void *sendbuf, int *sendcounts, int *displs,
    MPI_Datatype sendtype, void *recvbuf, int recvcount,
    MPI_Datatype recvtype, int root, MPI_Comm comm);
```

With MPI, you, the programmer, have to call the correct subroutine—the first if the source process is sending the same number of items to every destination process, the second if the source process is sending a different number of items to every destination process. You also have to supply the details about the data items' addresses, counts, types, and so on as subroutine arguments. Parallel Java's buffer objects take care of all this automatically.

Also, due to the separate buffer objects, Parallel Java's message passing operations are more flexible than MPI's. In MPI, all data for a message must come from or go into a single block of storage (designated by a single address, such as the `sendbuf` and `recvbuf` arguments in the MPI scatter subroutines). Parallel Java has no such limitation; different parts of a message can come from or go into different blocks of storage as implemented by the buffer object. In fact, this is crucial for sending and receiving matrices, which Java allocates in multiple storage blocks. In MPI it is possible to send and receive data of a nonprimitive type, such as a C struct; however, some intricate coding is required to make it happen. In Parallel Java, it is as easy to send a nonprimitive type (an object) as it is to send a primitive type like an `int`, simply by creating an ObjectBuf instead of an IntegerBuf.

Here are a few additional features of MPI. This is by no means a complete list; refer to the MPI standard for further information.

**Process topologies**. In some parallel programs, every process communicates with every other process, such as the program for Floyd's Algorithm in Chapter 25. In other parallel programs, this is not the case; each process communicates only with certain other processes. For example, in the pipelined antiproton motion program in Chapter 28, each process communicates only with its predecessor and its successor. The "process topology" is the pattern in which the processes communicate.

MPI has subroutines to declare the program's process topology. The MPI middleware can then use this information when deciding the backend node on which each process should execute. If processes are assigned to backend nodes such that the process topology matches the backend interconnection network topology, the program may experience better performance. Specifying the process topology probably won't make much difference for a program running on a cluster with a backend network consisting of a commodity Ethernet switch. However, it may make a difference for other backend network technologies, such as those that use mesh, torus, or hypercube networks.

Parallel Java at present does not support specifying process topologies.

**Multithreading support**. In MPI Version 1.1, the standard said nothing about supporting multithreaded programs. Thus, an MPI implementation could be designed with the assumption that the calling program was single-threaded—that is, without worrying about multiple-thread safety. Executing a multithreaded program with such an MPI implementation usually resulted in disaster. Although an MPI implementation *could* support multithreading, the standard did not say an implementation *had to* support multithreading.

MPI Version 2.0 added optional support for multithreaded programs. A multithreaded program must call `MPI_Init_thread()` instead of `MPI_Init()`. As an argument to `MPI_Init_thread()`, the program must specify the program's threading behavior, which can be one of the following:

- The program is single-threaded.
- The program is multithreaded, but only the "main" thread (the thread that called `MPI_Init_thread()`) will call MPI subroutines.
- The program is multithreaded, and multiple threads may call MPI subroutines, but only one thread at a time will ever call an MPI subroutine.
- The program is multithreaded, and multiple threads may call MPI subroutines concurrently.

However, an MPI implementation is still not required to support all these options. Thus, an attempt to initialize MPI with a certain level of thread support may fail. This means that a multithreaded MPI program is not guaranteed to run everywhere.

Parallel Java supports multithreaded message passing parallel programs with no restrictions.

**One-sided communication**. In MPI Version 2.0, each process can set up a "window" referring to a data item or group of data items by calling the `MPI_Win_create()` subroutine. Creating a window is a collective communication operation; every process in the program calls `MPI_Win_create()`, and every process becomes aware of every other process's window. Multiple separate windows can be created. Once a window is established, the following operations can be performed:

- One process A can call `MPI_Put()` to transfer data from itself into another process B's window. This is like a point-to-point message, except that instead of A doing a send and B doing a receive, A just does a put; B does not have to call anything.
- A can call `MPI_Get()` to transfer data into itself from B's window. This is like a point-to-point message, except that instead of A doing a receive and B doing a send, A just does a get; B does not have to call anything.
- A can call `MPI_Accumulate()` to combine its own data with data from B's window using a reduction operator, storing the result back into B's window. Again, B does not have to call anything.

Because only one of the two processes involved needs to call a subroutine to transfer the data, this capability is dubbed "one-sided" communication.

Parallel Java at present does not have such a capability.

**Parallel I/O**. MPI Version 2.0 includes platform-independent subroutines for file I/O, allowing MPI programs that access files to run unchanged on any operating system. An MPI program can also define "views" of a file, where the data items in the file are partitioned among the processes in a specified manner (Figure B.1) and each process sees just the data items in its own view. This supports the *parallel input files* and *parallel output files* patterns where each process reads or writes its own portion of a file in parallel with the other processes, rather than doing all file I/O in a single process (which can reduce performance). MPI implementations can also take advantage of special high-performance parallel file system hardware, if available.

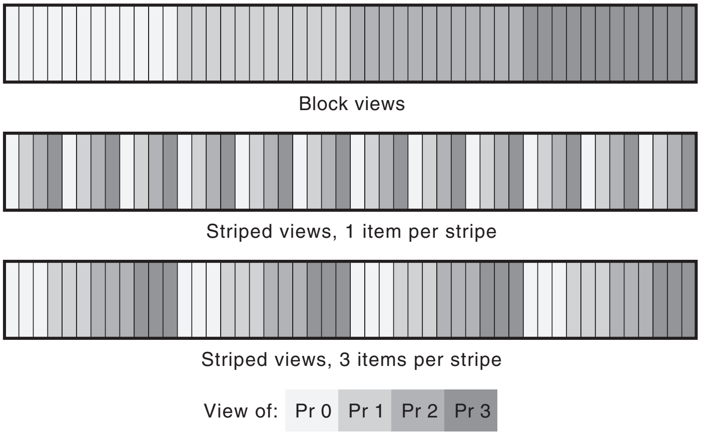

Block views
Striped views, 1 item per stripe
Striped views, 3 items per stripe
View of:
Pr 0
Pr 1
Pr 2
Pr 3

**Figure B.1** Parallel I/O file views in MPI Version 2.0

Parallel Java at present does not have a full-blown parallel file capability like MPI's. Some support for the *parallel input files* and *parallel output files* patterns is provided in the image classes in package edu.rit.image and the file classes in package edu.rit.io.

## B.3 MPI Performance

To get a sense of how a cluster parallel C program's performance compares with a cluster parallel Java program's performance, sequential and parallel versions of Floyd's Algorithm were implemented in C with MPI and in Java with Parallel Java. To make the comparison fair to Java, the C program included statements to do array index bounds checking, which Java does automatically, but C does not. To make the comparison fair to C, the C program was compiled with the highest level of optimization, which the JVM's JIT compiler does automatically.

The C and Java programs were compiled and run on the "paranoia" machine. This is an older and slower machine than the "tardis" machine that was used for cluster parallel program running-time measurements in the rest of this book. Each of the "paranoia" computer's 32 backend nodes has a 650-MHz Sun Microsystems UltraSPARC-IIe CPU chip and 1 GB of main memory. The backend machines are connected by a 100-Mbps switched Ethernet.

The C programs were compiled with the Sun C compiler at optimization level 5 (the highest possible) and used Sun's MPI library. The Java programs were run with the Sun JDK 1.5.0_15 HotSpot Server Virtual Machine. Table B.1 (at the end of the appendix) lists, and Figure B.2 plots, the C program's and the Java program's performance. The running times are for the calculation portion only. The program was run on distance matrices with the following numbers of vertices $n$ and problem sizes $N = n^3$:

| $n$ | $N$ | |
|---:|---:|---:|
| 1,000 | 1,000,000,000 | (1G) |
| 1,260 | 2,000,376,000 | (2G) |
| 1,590 | 4,019,679,000 | (4G) |
| 2,000 | 8,000,000,000 | (8G) |
| 2,520 | 16,003,008,000 | (16G) |
| 3,180 | 32,157,432,000 | (32G) |

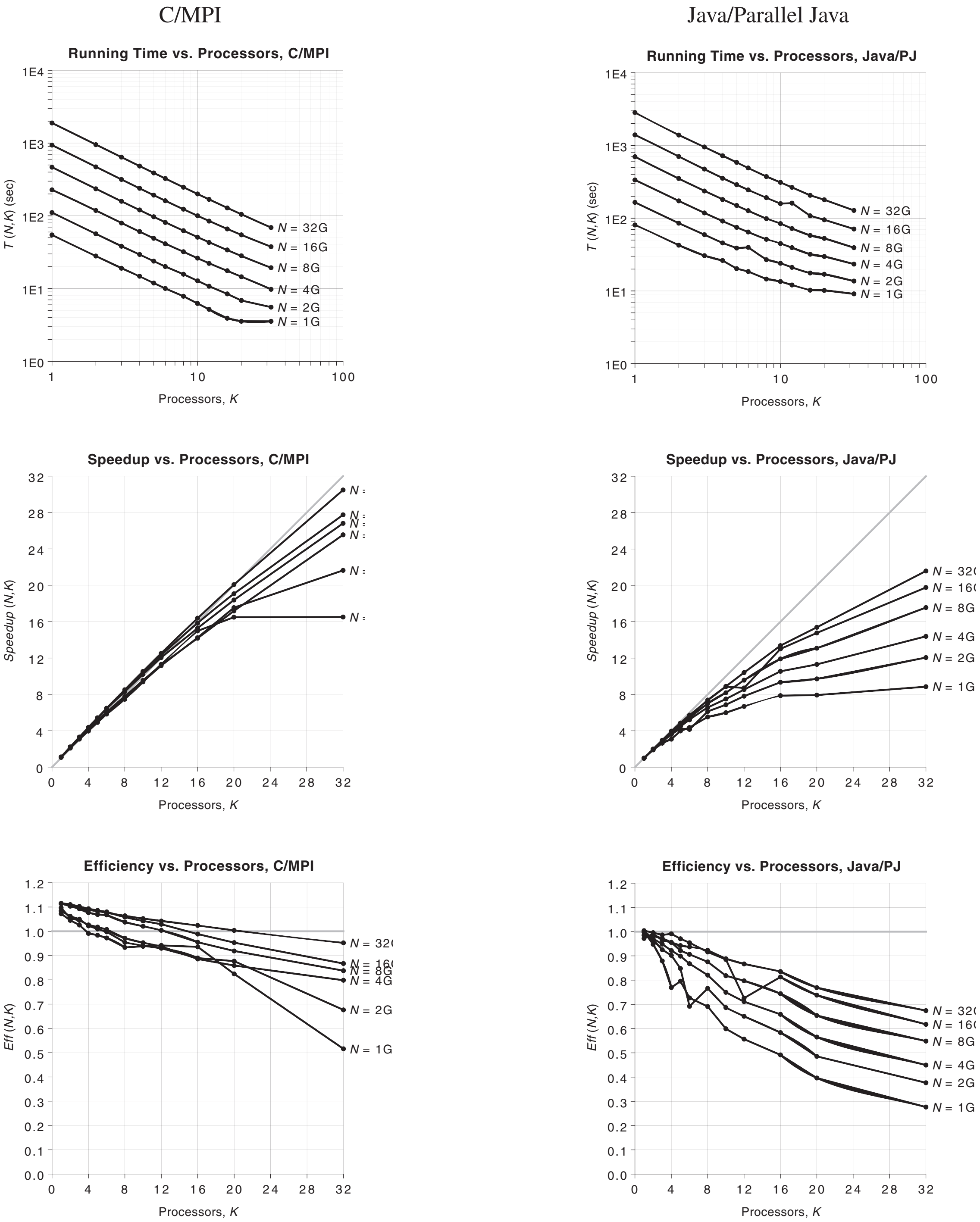


**Figure B.2** Floyd's Algorithm program running-time metrics

The Java program’s running times on one processor are about 50 percent higher than the C program’s running times. Also, the Java program’s speedups and efficiencies are not as high as the C program’s. This is likely due to a more efficient, platform-specific implementation of message passing in Sun’s MPI library. The Parallel Java Library’s platform-independent implementation results in additional overhead when sending and receiving messages.

As a further comparison between C and Java, sequential and cluster parallel versions of the Monte Carlo program for estimating π (see Chapter 26) were implemented in C with MPI and in Java with Parallel Java. Where the Floyd’s Algorithm program has a lot of communication (a broadcast at the top of every outer loop iteration), the π estimating program has little communication (one reduction at the end of the program). For the complete C source files, see the Parallel Java Library documentation (Javadoc). The source files are linked from the package summary page for package edu.rit.clu.monte.

The C and Java programs were compiled and run on the “paranoia” machine with *N* = 200 million, 500 million, 1 billion, 2 billion, 5 billion, and 10 billion darts. Table B.2 (at the end of the appendix) lists, and Figure B.3 plots, the C program’s and the Java program’s performance. The Java program’s running times are 30 percent smaller than the C program’s running times.

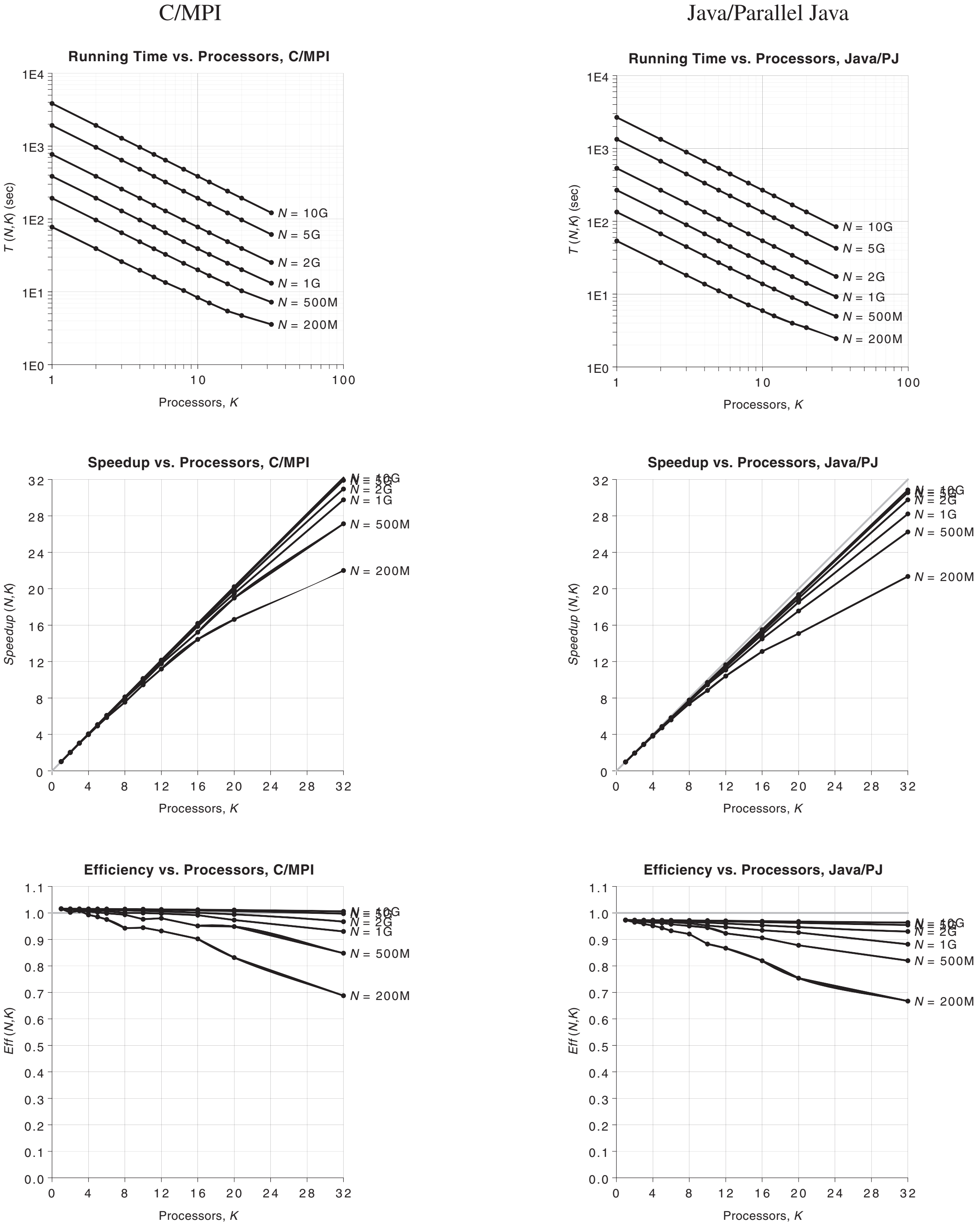


**Figure B.3** π estimating program running-time metrics

## B.4 For Further Information

On the official MPI standard:

- Message Passing Interface Forum Web site.
  http://www.mpi-forum.org/
- Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*. June 12, 1995. (MPI Version 1.1)
  http://www.mpi-forum.org/docs/mpi-11.ps
- Message Passing Interface Forum. *MPI-2: Extensions to the Message-Passing Interface*. July 18, 1997. (MPI Version 2.0)
  http://www.mpi-forum.org/docs/mpi-20.ps

Textbooks on parallel programming with MPI:

- B. Wilkinson and M. Allen. *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers, Second Edition.* Prentice-Hall, 2005.
- M. Quinn. *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill, 2004.
- G. Karniadakis and R. Kirby II. *Parallel Scientific Computing in C++ and MPI: A Seamless Approach to Parallel Algorithms and Their Implementation*. Cambridge University Press, 2003.
- W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing Interface, Second Edition.* MIT Press, 1999.
- P. Pacheco. *A User's Guide to MPI.* March 30, 1998.
  ftp://math.usfca.edu/pub/MPI/mpi.guide.ps.Z
- P. Pacheco. *Parallel Programming with MPI*. Morgan Kaufmann, 1997.

**Table B.1** Floyd's Algorithm program running-time metrics

| C/MPI | | | | | | Java/Parallel Java | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| *N* | *K* | *T* | *Spdup* | *Eff* | *EDSF* | *N* | *K* | *T* | *Spdup* | *Eff* | *EDSF* |
| 1G | seq | 58639 | | | | 1G | seq | 80661 | | | |
| 1G | 1 | 54685 | 1.072 | 1.072 | | 1G | 1 | 80745 | 0.999 | 0.999 | |
| 1G | 2 | 28056 | 2.090 | 1.045 | 0.026 | 1G | 2 | 42560 | 1.895 | 0.948 | 0.054 |
| 1G | 3 | 19054 | 3.078 | 1.026 | 0.023 | 1G | 3 | 30574 | 2.638 | 0.879 | 0.068 |
| 1G | 4 | 14789 | 3.965 | 0.991 | 0.027 | 1G | 4 | 26208 | 3.078 | 0.769 | 0.099 |
| 1G | 5 | 11921 | 4.919 | 0.984 | 0.022 | 1G | 5 | 20272 | 3.979 | 0.796 | 0.064 |
| 1G | 6 | 10050 | 5.835 | 0.972 | 0.021 | 1G | 6 | 18494 | 4.361 | 0.727 | 0.075 |
| 1G | 8 | 7852 | 7.468 | 0.934 | 0.021 | 1G | 8 | 14586 | 5.530 | 0.691 | 0.064 |
| 1G | 10 | 6250 | 9.382 | 0.938 | 0.016 | 1G | 10 | 13457 | 5.994 | 0.599 | 0.074 |
| 1G | 12 | 5189 | 11.301 | 0.942 | 0.013 | 1G | 12 | 12074 | 6.681 | 0.557 | 0.072 |
| 1G | 16 | 3916 | 14.974 | 0.936 | 0.010 | 1G | 16 | 10251 | 7.869 | 0.492 | 0.069 |
| 1G | 20 | 3556 | 16.490 | 0.825 | 0.016 | 1G | 20 | 10173 | 7.929 | 0.396 | 0.080 |
| 1G | 32 | 3552 | 16.509 | 0.516 | 0.035 | 1G | 32 | 9111 | 8.853 | 0.277 | 0.084 |
| 2G | seq | 120397 | | | | 2G | seq | 164884 | | | |
| 2G | 1 | 111047 | 1.084 | 1.084 | | 2G | 1 | 165191 | 0.998 | 0.998 | |
| 2G | 2 | 56695 | 2.124 | 1.062 | 0.021 | 2G | 2 | 85423 | 1.930 | 0.965 | 0.034 |
| 2G | 3 | 38223 | 3.150 | 1.050 | 0.016 | 2G | 3 | 59377 | 2.777 | 0.926 | 0.039 |
| 2G | 4 | 29444 | 4.089 | 1.022 | 0.020 | 2G | 4 | 45682 | 3.609 | 0.902 | 0.035 |
| 2G | 5 | 23849 | 5.048 | 1.010 | 0.018 | 2G | 5 | 38866 | 4.242 | 0.848 | 0.044 |
| 2G | 6 | 20113 | 5.986 | 0.998 | 0.017 | 2G | 6 | 39700 | 4.153 | 0.692 | 0.088 |
| 2G | 8 | 15770 | 7.635 | 0.954 | 0.019 | 2G | 8 | 26902 | 6.129 | 0.766 | 0.043 |
| 2G | 10 | 12801 | 9.405 | 0.941 | 0.017 | 2G | 10 | 24011 | 6.867 | 0.687 | 0.050 |
| 2G | 12 | 10789 | 11.159 | 0.930 | 0.015 | 2G | 12 | 21110 | 7.811 | 0.651 | 0.048 |
| 2G | 16 | 8459 | 14.233 | 0.890 | 0.015 | 2G | 16 | 17653 | 9.340 | 0.584 | 0.047 |
| 2G | 20 | 6864 | 17.540 | 0.877 | 0.012 | 2G | 20 | 16977 | 9.712 | 0.486 | 0.056 |
| 2G | 32 | 5564 | 21.639 | 0.676 | 0.019 | 2G | 32 | 13676 | 12.056 | 0.377 | 0.053 |
| 4G | seq | 249909 | | | | 4G | seq | 336484 | | | |
| 4G | 1 | 227871 | 1.097 | 1.097 | | 4G | 1 | 335668 | 1.002 | 1.002 | |
| 4G | 2 | 118641 | 2.106 | 1.053 | 0.041 | 4G | 2 | 173215 | 1.943 | 0.971 | 0.032 |
| 4G | 3 | 79704 | 3.135 | 1.045 | 0.025 | 4G | 3 | 118248 | 2.846 | 0.949 | 0.028 |
| 4G | 4 | 60875 | 4.105 | 1.026 | 0.023 | 4G | 4 | 91237 | 3.688 | 0.922 | 0.029 |
| 4G | 5 | 49126 | 5.087 | 1.017 | 0.019 | 4G | 5 | 74844 | 4.496 | 0.899 | 0.029 |
| 4G | 6 | 41344 | 6.045 | 1.007 | 0.018 | 4G | 6 | 64617 | 5.207 | 0.868 | 0.031 |
| 4G | 8 | 32152 | 7.773 | 0.972 | 0.018 | 4G | 8 | 51247 | 6.566 | 0.821 | 0.032 |
| 4G | 10 | 26226 | 9.529 | 0.953 | 0.017 | 4G | 10 | 44875 | 7.498 | 0.750 | 0.037 |
| 4G | 12 | 22217 | 11.249 | 0.937 | 0.015 | 4G | 12 | 39419 | 8.536 | 0.711 | 0.037 |
| 4G | 16 | 17632 | 14.174 | 0.886 | 0.016 | 4G | 16 | 31913 | 10.544 | 0.659 | 0.035 |
| 4G | 20 | 14547 | 17.179 | 0.859 | 0.015 | 4G | 20 | 29764 | 11.305 | 0.565 | 0.041 |
| 4G | 32 | 9783 | 25.545 | 0.798 | 0.012 | 4G | 32 | 23382 | 14.391 | 0.450 | 0.040 |

**Table B.1** Floyd’s Algorithm program running-time metrics (cont.)

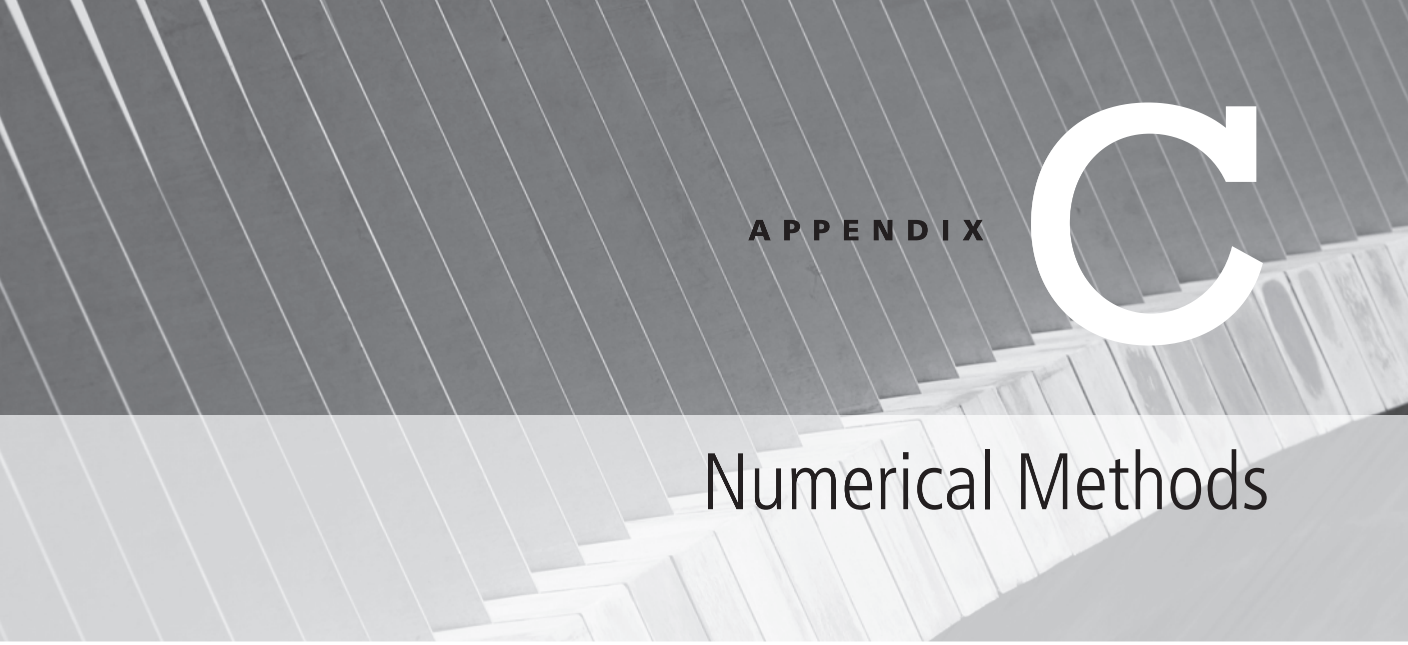
| C/MPI | | | | | | Java/Parallel Java | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| *N* | *K* | *T* | *Spdup* | *Eff* | *EDSF* | *N* | *K* | *T* | *Spdup* | *Eff* | *EDSF* |
| 8G | seq | 519582 | | | | 8G | seq | 690626 | | | |
| 8G | 1 | 466349 | 1.114 | 1.114 | | 8G | 1 | 700838 | 0.985 | 0.985 | |
| 8G | 2 | 235456 | 2.207 | 1.103 | 0.010 | 8G | 2 | 351825 | 1.963 | 0.981 | 0.004 |
| 8G | 3 | 158512 | 3.278 | 1.093 | 0.010 | 8G | 3 | 236787 | 2.917 | 0.972 | 0.007 |
| 8G | 4 | 120645 | 4.307 | 1.077 | 0.012 | 8G | 4 | 180722 | 3.821 | 0.955 | 0.010 |
| 8G | 5 | 97198 | 5.346 | 1.069 | 0.011 | 8G | 5 | 149818 | 4.610 | 0.922 | 0.017 |
| 8G | 6 | 81126 | 6.405 | 1.067 | 0.009 | 8G | 6 | 126998 | 5.438 | 0.906 | 0.017 |
| 8G | 8 | 62583 | 8.302 | 1.038 | 0.011 | 8G | 8 | 98606 | 7.004 | 0.875 | 0.018 |
| 8G | 10 | 50903 | 10.207 | 1.021 | 0.010 | 8G | 10 | 84401 | 8.183 | 0.818 | 0.023 |
| 8G | 12 | 43133 | 12.046 | 1.004 | 0.010 | 8G | 12 | 72171 | 9.569 | 0.797 | 0.021 |
| 8G | 16 | 34005 | 15.280 | 0.955 | 0.011 | 8G | 16 | 57995 | 11.908 | 0.744 | 0.022 |
| 8G | 20 | 28270 | 18.379 | 0.919 | 0.011 | 8G | 20 | 52780 | 13.085 | 0.654 | 0.027 |
| 8G | 32 | 19381 | 26.809 | 0.838 | 0.011 | 8G | 32 | 39336 | 17.557 | 0.549 | 0.026 |
| 16G | seq | 1045915 | | | | 16G | seq | 1404550 | | | |
| 16G | 1 | 937904 | 1.115 | 1.115 | | 16G | 1 | 1398564 | 1.004 | 1.004 | |
| 16G | 2 | 471136 | 2.220 | 1.110 | 0.005 | 16G | 2 | 705170 | 1.992 | 0.996 | 0.008 |
| 16G | 3 | 316234 | 3.307 | 1.102 | 0.006 | 16G | 3 | 474557 | 2.960 | 0.987 | 0.009 |
| 16G | 4 | 239252 | 4.372 | 1.093 | 0.007 | 16G | 4 | 354286 | 3.964 | 0.991 | 0.004 |
| 16G | 5 | 192635 | 5.430 | 1.086 | 0.007 | 16G | 5 | 289351 | 4.854 | 0.971 | 0.009 |
| 16G | 6 | 161416 | 6.480 | 1.080 | 0.007 | 16G | 6 | 245271 | 5.727 | 0.954 | 0.010 |
| 16G | 8 | 123541 | 8.466 | 1.058 | 0.008 | 16G | 8 | 191650 | 7.329 | 0.916 | 0.014 |
| 16G | 10 | 100324 | 10.425 | 1.043 | 0.008 | 16G | 10 | 158653 | 8.853 | 0.885 | 0.015 |
| 16G | 12 | 84700 | 12.348 | 1.029 | 0.008 | 16G | 12 | 161202 | 8.713 | 0.726 | 0.035 |
| 16G | 16 | 66128 | 15.817 | 0.989 | 0.009 | 16G | 16 | 107968 | 13.009 | 0.813 | 0.016 |
| 16G | 20 | 54858 | 19.066 | 0.953 | 0.009 | 16G | 20 | 95143 | 14.763 | 0.738 | 0.019 |
| 16G | 32 | 37692 | 27.749 | 0.867 | 0.009 | 16G | 32 | 71054 | 19.767 | 0.618 | 0.020 |
| 32G | seq | 2106085 | | | | 32G | seq | 2761066 | | | |
| 32G | 1 | 1891309 | 1.114 | 1.114 | | 32G | 1 | 2840370 | 0.972 | 0.972 | |
| 32G | 2 | 951322 | 2.214 | 1.107 | 0.006 | 32G | 2 | 1391888 | 1.984 | 0.992 | -0.020 |
| 32G | 3 | 641446 | 3.283 | 1.094 | 0.009 | 32G | 3 | 954381 | 2.893 | 0.964 | 0.004 |
| 32G | 4 | 483654 | 4.355 | 1.089 | 0.008 | 32G | 4 | 722122 | 3.824 | 0.956 | 0.006 |
| 32G | 5 | 389189 | 5.411 | 1.082 | 0.007 | 32G | 5 | 585715 | 4.714 | 0.943 | 0.008 |
| 32G | 6 | 326248 | 6.455 | 1.076 | 0.007 | 32G | 6 | 491069 | 5.623 | 0.937 | 0.007 |
| 32G | 8 | 247466 | 8.511 | 1.064 | 0.007 | 32G | 8 | 373809 | 7.386 | 0.923 | 0.008 |
| 32G | 10 | 200135 | 10.523 | 1.052 | 0.006 | 32G | 10 | 310758 | 8.885 | 0.888 | 0.010 |
| 32G | 12 | 168425 | 12.505 | 1.042 | 0.006 | 32G | 12 | 265366 | 10.405 | 0.867 | 0.011 |
| 32G | 16 | 128527 | 16.386 | 1.024 | 0.006 | 32G | 16 | 206515 | 13.370 | 0.836 | 0.011 |
| 32G | 20 | 104935 | 20.070 | 1.004 | 0.006 | 32G | 20 | 179458 | 15.386 | 0.769 | 0.014 |
| 32G | 32 | 69109 | 30.475 | 0.952 | 0.005 | 32G | 32 | 127957 | 21.578 | 0.674 | 0.014 |

**Table B.2** $\pi$ estimating program running-time metrics

| C/MPI | | | | | | Java/Parallel Java | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| *N* | *K* | *T* | *Spdup* | *Eff* | *EDSF* | *N* | *K* | *T* | *Spdup* | *Eff* | *EDSF* |
| 200M | seq | 78510 | | | | 200M | seq | 52221 | | | |
| 200M | 1 | 77330 | 1.015 | 1.015 | | 200M | 1 | 53654 | 0.973 | 0.973 | |
| 200M | 2 | 39171 | 2.004 | 1.002 | 0.013 | 200M | 2 | 27061 | 1.930 | 0.965 | 0.009 |
| 200M | 3 | 25953 | 3.025 | 1.008 | 0.003 | 200M | 3 | 18150 | 2.877 | 0.959 | 0.007 |
| 200M | 4 | 19773 | 3.971 | 0.993 | 0.008 | 200M | 4 | 13723 | 3.805 | 0.951 | 0.008 |
| 200M | 5 | 15937 | 4.926 | 0.985 | 0.008 | 200M | 5 | 11071 | 4.717 | 0.943 | 0.008 |
| 200M | 6 | 13415 | 5.852 | 0.975 | 0.008 | 200M | 6 | 9333 | 5.595 | 0.933 | 0.009 |
| 200M | 8 | 10413 | 7.540 | 0.942 | 0.011 | 200M | 8 | 7092 | 7.363 | 0.920 | 0.008 |
| 200M | 10 | 8313 | 9.444 | 0.944 | 0.008 | 200M | 10 | 5916 | 8.827 | 0.883 | 0.011 |
| 200M | 12 | 7021 | 11.182 | 0.932 | 0.008 | 200M | 12 | 5020 | 10.403 | 0.867 | 0.011 |
| 200M | 16 | 5439 | 14.435 | 0.902 | 0.008 | 200M | 16 | 3983 | 13.111 | 0.819 | 0.013 |
| 200M | 20 | 4721 | 16.630 | 0.831 | 0.012 | 200M | 20 | 3464 | 15.075 | 0.754 | 0.015 |
| 200M | 32 | 3568 | 22.004 | 0.688 | 0.015 | 200M | 32 | 2446 | 21.350 | 0.667 | 0.015 |
| 500M | seq | 195540 | | | | 500M | seq | 130056 | | | |
| 500M | 1 | 192530 | 1.016 | 1.016 | | 500M | 1 | 133678 | 0.973 | 0.973 | |
| 500M | 2 | 96644 | 2.023 | 1.012 | 0.004 | 500M | 2 | 67066 | 1.939 | 0.970 | 0.003 |
| 500M | 3 | 64614 | 3.026 | 1.009 | 0.003 | 500M | 3 | 44818 | 2.902 | 0.967 | 0.003 |
| 500M | 4 | 48631 | 4.021 | 1.005 | 0.003 | 500M | 4 | 33743 | 3.854 | 0.964 | 0.003 |
| 500M | 5 | 39002 | 5.014 | 1.003 | 0.003 | 500M | 5 | 27038 | 4.810 | 0.962 | 0.003 |
| 500M | 6 | 32642 | 5.990 | 0.998 | 0.003 | 500M | 6 | 22641 | 5.744 | 0.957 | 0.003 |
| 500M | 8 | 24607 | 7.947 | 0.993 | 0.003 | 500M | 8 | 17098 | 7.607 | 0.951 | 0.003 |
| 500M | 10 | 20028 | 9.763 | 0.976 | 0.004 | 500M | 10 | 13771 | 9.444 | 0.944 | 0.003 |
| 500M | 12 | 16630 | 11.758 | 0.980 | 0.003 | 500M | 12 | 11742 | 11.076 | 0.923 | 0.005 |
| 500M | 16 | 12849 | 15.218 | 0.951 | 0.005 | 500M | 16 | 8973 | 14.494 | 0.906 | 0.005 |
| 500M | 20 | 10306 | 18.973 | 0.949 | 0.004 | 500M | 20 | 7408 | 17.556 | 0.878 | 0.006 |
| 500M | 32 | 7206 | 27.136 | 0.848 | 0.006 | 500M | 32 | 4957 | 26.237 | 0.820 | 0.006 |
| 1G | seq | 390885 | | | | 1G | seq | 259812 | | | |
| 1G | 1 | 384801 | 1.016 | 1.016 | | 1G | 1 | 267048 | 0.973 | 0.973 | |
| 1G | 2 | 192801 | 2.027 | 1.014 | 0.002 | 1G | 2 | 133726 | 1.943 | 0.971 | 0.002 |
| 1G | 3 | 128727 | 3.037 | 1.012 | 0.002 | 1G | 3 | 89288 | 2.910 | 0.970 | 0.002 |
| 1G | 4 | 96671 | 4.043 | 1.011 | 0.002 | 1G | 4 | 67079 | 3.873 | 0.968 | 0.002 |
| 1G | 5 | 77494 | 5.044 | 1.009 | 0.002 | 1G | 5 | 53768 | 4.832 | 0.966 | 0.002 |
| 1G | 6 | 64701 | 6.041 | 1.007 | 0.002 | 1G | 6 | 44878 | 5.789 | 0.965 | 0.002 |
| 1G | 8 | 48855 | 8.001 | 1.000 | 0.002 | 1G | 8 | 33800 | 7.687 | 0.961 | 0.002 |
| 1G | 10 | 39096 | 9.998 | 1.000 | 0.002 | 1G | 10 | 27276 | 9.525 | 0.953 | 0.002 |
| 1G | 12 | 32648 | 11.973 | 0.998 | 0.002 | 1G | 12 | 22873 | 11.359 | 0.947 | 0.003 |
| 1G | 16 | 24638 | 15.865 | 0.992 | 0.002 | 1G | 16 | 17377 | 14.951 | 0.934 | 0.003 |
| 1G | 20 | 20082 | 19.464 | 0.973 | 0.002 | 1G | 20 | 14028 | 18.521 | 0.926 | 0.003 |
| 1G | 32 | 13133 | 29.764 | 0.930 | 0.003 | 1G | 32 | 9207 | 28.219 | 0.882 | 0.003 |

**Table B.2** π estimating program running-time metrics (cont.)

| C/MPI | | | | | | Java/Parallel Java | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| *N* | *K* | *T* | *Spdup* | *Eff* | *EDSF* | *N* | *K* | *T* | *Spdup* | *Eff* | *EDSF* |
| 2G | seq | 781538 | | | | 2G | seq | 519331 | | | |
| 2G | 1 | 769499 | 1.016 | 1.016 | | 2G | 1 | 533687 | 0.973 | 0.973 | |
| 2G | 2 | 385141 | 2.029 | 1.015 | 0.001 | 2G | 2 | 267071 | 1.945 | 0.972 | 0.001 |
| 2G | 3 | 256898 | 3.042 | 1.014 | 0.001 | 2G | 3 | 178189 | 2.914 | 0.971 | 0.001 |
| 2G | 4 | 192821 | 4.053 | 1.013 | 0.001 | 2G | 4 | 133785 | 3.882 | 0.970 | 0.001 |
| 2G | 5 | 154371 | 5.063 | 1.013 | 0.001 | 2G | 5 | 107134 | 4.847 | 0.969 | 0.001 |
| 2G | 6 | 128784 | 6.069 | 1.011 | 0.001 | 2G | 6 | 89345 | 5.813 | 0.969 | 0.001 |
| 2G | 8 | 96748 | 8.078 | 1.010 | 0.001 | 2G | 8 | 67135 | 7.736 | 0.967 | 0.001 |
| 2G | 10 | 77556 | 10.077 | 1.008 | 0.001 | 2G | 10 | 53921 | 9.631 | 0.963 | 0.001 |
| 2G | 12 | 64729 | 12.074 | 1.006 | 0.001 | 2G | 12 | 45061 | 11.525 | 0.960 | 0.001 |
| 2G | 16 | 48780 | 16.022 | 1.001 | 0.001 | 2G | 16 | 34067 | 15.244 | 0.953 | 0.001 |
| 2G | 20 | 39309 | 19.882 | 0.994 | 0.001 | 2G | 20 | 27423 | 18.938 | 0.947 | 0.001 |
| 2G | 32 | 25253 | 30.948 | 0.967 | 0.002 | 2G | 32 | 17450 | 29.761 | 0.930 | 0.001 |
| 5G | seq | 1953738 | | | | 5G | seq | 1297842 | | | |
| 5G | 1 | 1923311 | 1.016 | 1.016 | | 5G | 1 | 1333845 | 0.973 | 0.973 | |
| 5G | 2 | 962138 | 2.031 | 1.015 | 0.001 | 5G | 2 | 667187 | 1.945 | 0.973 | 0.000 |
| 5G | 3 | 641593 | 3.045 | 1.015 | 0.000 | 5G | 3 | 444931 | 2.917 | 0.972 | 0.000 |
| 5G | 4 | 481353 | 4.059 | 1.015 | 0.000 | 5G | 4 | 333875 | 3.887 | 0.972 | 0.000 |
| 5G | 5 | 385199 | 5.072 | 1.014 | 0.000 | 5G | 5 | 267181 | 4.858 | 0.972 | 0.000 |
| 5G | 6 | 321104 | 6.084 | 1.014 | 0.000 | 5G | 6 | 222722 | 5.827 | 0.971 | 0.000 |
| 5G | 8 | 241142 | 8.102 | 1.013 | 0.000 | 5G | 8 | 167122 | 7.766 | 0.971 | 0.000 |
| 5G | 10 | 192905 | 10.128 | 1.013 | 0.000 | 5G | 10 | 133862 | 9.695 | 0.970 | 0.000 |
| 5G | 12 | 160890 | 12.143 | 1.012 | 0.000 | 5G | 12 | 111718 | 11.617 | 0.968 | 0.000 |
| 5G | 16 | 120945 | 16.154 | 1.010 | 0.000 | 5G | 16 | 84079 | 15.436 | 0.965 | 0.001 |
| 5G | 20 | 97025 | 20.136 | 1.007 | 0.000 | 5G | 20 | 67388 | 19.259 | 0.963 | 0.001 |
| 5G | 32 | 61219 | 31.914 | 0.997 | 0.001 | 5G | 32 | 42489 | 30.545 | 0.955 | 0.001 |
| 10G | seq | 3907132 | | | | 10G | seq | 2595385 | | | |
| 10G | 1 | 3846674 | 1.016 | 1.016 | | 10G | 1 | 2667343 | 0.973 | 0.973 | |
| 10G | 2 | 1924194 | 2.031 | 1.015 | 0.000 | 10G | 2 | 1333923 | 1.946 | 0.973 | 0.000 |
| 10G | 3 | 1282690 | 3.046 | 1.015 | 0.000 | 10G | 3 | 889483 | 2.918 | 0.973 | 0.000 |
| 10G | 4 | 962154 | 4.061 | 1.015 | 0.000 | 10G | 4 | 667150 | 3.890 | 0.973 | 0.000 |
| 10G | 5 | 769882 | 5.075 | 1.015 | 0.000 | 10G | 5 | 533876 | 4.861 | 0.972 | 0.000 |
| 10G | 6 | 641632 | 6.089 | 1.015 | 0.000 | 10G | 6 | 444936 | 5.833 | 0.972 | 0.000 |
| 10G | 8 | 481354 | 8.117 | 1.015 | 0.000 | 10G | 8 | 333867 | 7.774 | 0.972 | 0.000 |
| 10G | 10 | 385291 | 10.141 | 1.014 | 0.000 | 10G | 10 | 267243 | 9.712 | 0.971 | 0.000 |
| 10G | 12 | 321387 | 12.157 | 1.013 | 0.000 | 10G | 12 | 222924 | 11.642 | 0.970 | 0.000 |
| 10G | 16 | 241244 | 16.196 | 1.012 | 0.000 | 10G | 16 | 167470 | 15.498 | 0.969 | 0.000 |
| 10G | 20 | 193205 | 20.223 | 1.011 | 0.000 | 10G | 20 | 134106 | 19.353 | 0.968 | 0.000 |
| 10G | 32 | 121374 | 32.191 | 1.006 | 0.000 | 10G | 32 | 84158 | 30.839 | 0.964 | 0.000 |



# APPENDIX C

# Numerical Methods

in which we survey several numerical methods used in this book, including log-log plots; power functions; power function curve fitting; linear regression; general linear least-squares curve fitting; and quadratic and cubic equations

# APPENDIX C Numerical Methods

## C.1 Log-Log Plots

A regular plot uses a **linear scale** for each axis. Each tick mark's value is obtained by adding a fixed amount to the previous tick mark's value. Putting it another way, the tick mark values form an arithmetic series. However, for some data sets, a plot with linear scales isn't very helpful.

Consider the running-time data for the AES key search program from Chapter 9. Figure C.1 shows this data set on a regular plot. Because the *T* values for *N* = 512M are so large, only the first two or three curves show up well on the plot. The other curves, with much smaller *T* values, are all compressed together at the bottom of the plot. This makes it difficult to see what the curves look like or whether there are any anomalies in the data.

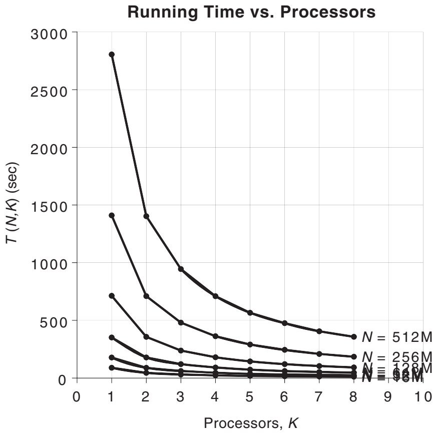


**Figure C.1** FindKeySeq/FindKeySmp3 *T* vs. *K*

Suppose we keep the linear scales, but rather than plotting the data itself, we plot the *logarithm* of the data. That is, we plot (log *T*) versus (log *K*) instead of *T* versus *K*. Figure C.2 shows the result. Now all the curves are visible and well separated. Furthermore, the curves show up as straight lines, an observation we will return to shortly.

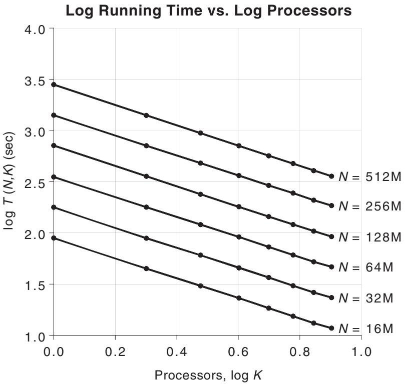


**Figure C.2** FindKeySeq/FindKeySmp3 (log *T*) vs. (log *K*)

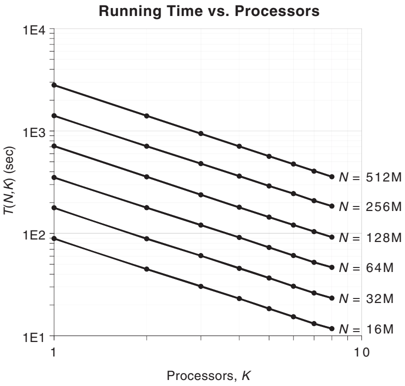


**Figure C.3** FindKeySeq/FindKeySmp3 *T* vs. *K*, log-log plot

The logarithm we are using is the *base-10 logarithm,* sometimes written as "$\log_{10}$." This is not the same as the natural logarithm or logarithm to the base e, sometimes written as "ln." On calculators, the base-10 logarithm key is usually labeled "log," and the natural logarithm key is usually labeled "ln." In Java, the `Math.log()` function computes the natural logarithm and the `Math.log10()` function computes the base-10 logarithm. You can also compute the base-10 logarithm of *x* as (ln *x* / ln 10).

While the *curves* show up nicely when we plot the logarithm of the data, the *tick marks* on the axes do not turn out as nicely—they show the logarithms of the data values, which are not as meaningful as the data values themselves. Suppose we continue to plot the logarithms of the data values, but we mark the axes with the actual data values. Figure C.3 shows the result. By convention, the major divisions correspond to integer powers of 10—$1 \times 10^1$ (`1E1`), $1 \times 10^2$ (`1E2`), and so on. Within each major division, the minor divisions correspond to integer mantissas—$2 \times 10^1$, $3 \times 10^1$, . . . , $9 \times 10^1$, and so on. The minor divisions are not evenly spaced because the logarithm function is nonlinear.

The scale used for each axis in a plot like Figure C.3 is called a **logarithmic scale**. Each major tick mark's value is obtained by *multiplying* the previous major tick mark's value by a fixed amount. Putting it another way, the major tick mark values form a *geometric* series. Because the horizontal axis uses a logarithmic (log) scale and the vertical axis also uses a log scale, the whole thing is called a **log-log plot**.

## C.2 Power Functions on a Log-Log Plot

Let's look at how a **power function** appears on a log-log plot. A power function $y(x)$ is one of the form

$$y = ax^b \tag{C.1}$$

where *a* and *b* are constants. In computer science, power functions are interesting because many algorithms have a polynomial complexity. For example, an $O(n^2)$ algorithm's running time is less than or equal to some constant times $n^2$, which is a power function with $b = 2$. A parallel program's running time *T* for a given problem size is ideally supposed to be inversely proportional to the number of processors *K*; that is, *T* is some constant times $1/K$ ($b = -1$).

On a log-log plot, we plot a data point $(x,y)$ in alternate coordinates $(u,v)$, where $u = \log x$ and $v = \log y$. Taking the logarithm of both sides of Equation C.1 gives the following:

$$\log y = \log a + b \log x \tag{C.2}$$

Expressing this in terms of $u$ and $v$ gives

$$v = c + bu \tag{C.3}$$

where $c = \log a$. This is the equation for a straight line in $(u,v)$ coordinates, where $b$ is the slope and $c$ is the intercept.

Consequently, a power function shows up as a straight line on a log-log plot, with slope equal to the power of $x$. Figure C.4 plots several power functions with the same multiplier $a = 1$ and different exponents. The multiplier shifts the curve up or down without changing its slope. Figure C.5 plots several power functions with different multipliers and the same exponent $b = 2$.

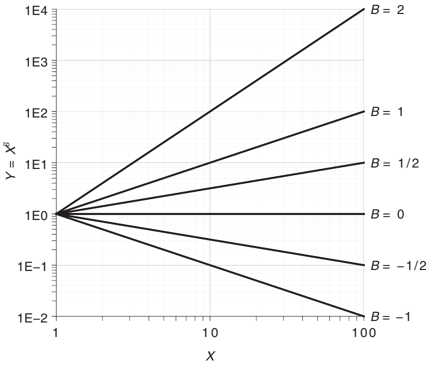


**Figure C.4** Log-log plots of $y = x^b$ for various exponents

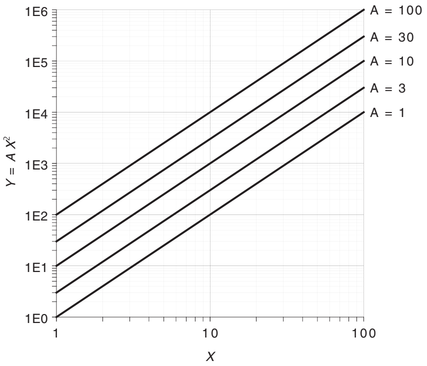


**Figure C.5** Log-log plots of $y = ax^2$ for various multipliers

Thus, a log-log plot is ideal for eyeballing whether one quantity is proportional to a power of another quantity. Just look for a straight line and see if it has the right slope.

## C.3 Power Function Curve Fitting

An experiment yields a series of $(x,y)$ measurements where $y$ is supposed to be a power function of $x$, $y = ax^b$, with a predetermined exponent $b$. Due to random measurement errors, the data points do not fall precisely on a straight line on a log-log plot. We want to determine the coefficient $a$ that gives the closest fit between the **model function** $y = ax^b$ and the data series $(x,y)$. This is the **power function curve-fitting** problem.

As an example, in Chapter 25 we needed to fit the following data to the model $y = ax^3$, where $x$ was the number of vertices in a distance matrix and $y$ was the running time (sec) for a sequential version of Floyd's Algorithm:

| $x$ | $y$ |
|---:|---:|
| 2000 | 67.942 |
| 2520 | 136.506 |
| 3180 | 269.528 |
| 4000 | 589.131 |
| 5040 | 1182.404 |
| 6360 | 2474.565 |

Rearranging Equation C.2 gives a formula for (log $a$) in terms of the predetermined exponent $b$ and the data values $(x,y)$:

$$\log\ a = \log\ y - b\ \log\ x \tag{C.4}$$

Plugging different data values into Equation C.4 gives different values for (log $a$). In the example, with $b = 3$, the values are the following:

| $x$ | $y$ | log $a$ |
|---:|---:|---:|
| 2000 | 67.942 | −8.071 |
| 2520 | 136.506 | −8.069 |
| 3180 | 269.528 | −8.077 |
| 4000 | 589.131 | −8.036 |
| 5040 | 1182.404 | −8.035 |
| 6360 | 2474.565 | −8.017 |

If the data fell on a straight line on a log-log plot, all the (log $a$) values would be the same, namely the intercept of the line. Due to random measurement errors, the (log $a$) values are close, but not identical. To get the best fit between the straight line (the model function) and the data, we will take the intercept to be the *median* of the (log $a$) values. To find the median, sort the list of values; if there are an odd number of values, then the median is the middle value; if there are an even number of values, then the median is halfway between the two middle values. In the example, the median (log $a$) value is –8.053. Raising 10 to the power of (log $a$) gives $a = 8.86 \times 10^{-9}$. The model function is then $y = 8.86 \times 10^{-9}\ x^3$.

Figure C.6 shows the example data and the fitted model function. Two of the six data points—corresponding to the two middle vales in the sorted list—are closest to the straight line, and the straight line splits the difference between them.

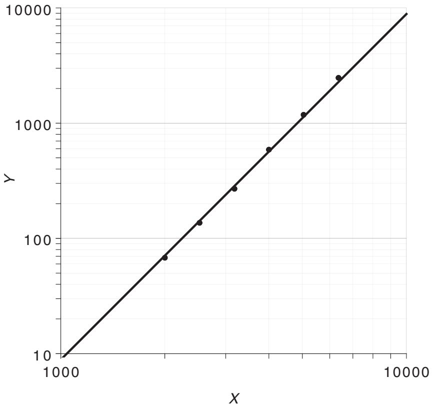


**Figure C.6** Example of a power function curve fit

How is this model function the "best" fit to the data? Define the **total absolute deviation** *D* to be the sum of the absolute differences between the actual *y* value and the model function for each data point, *in logarithmic coordinates*:

$$D = \sum_i \left| \log y_i - (\log a + b \log x_i) \right| \tag{C.5}$$

Then choosing (log *a*) as just described will minimize *D*. That is, the best curve fit is the one that minimizes the total absolute deviation.

## C.4 Linear Regression

We are given a series of data points ($x_i$, $y_i$), $i = 1, 2, ..., M$. We want to find a straight line of the form

$$y = a + bx \tag{C.6}$$

that is the best fit to the data points. Equation C.6 is a **model** of the data, and the coefficients *a* and *b* are the **model parameters**. Fitting a model to the data is called "regression," and when the model is a straight line the process is called **linear regression**.

First, compute the means of the *x* and *y* values:

$$\overline{x} = \frac{1}{M} \sum_{i=1}^{M} x_i \tag{C.7}$$

$$\overline{y} = \frac{1}{M} \sum_{i=1}^{M} y_i \tag{C.8}$$

Then, the model parameters are the following:

$$b = \frac{\sum_{i=1}^{M} (x_i - \overline{x}) y_i}{\sum_{i=1}^{M} (x_i - \overline{x})^2} \tag{C.9}$$

$$a = \overline{y} - b\,\overline{x} \tag{C.10}$$

The model function (C.6) gives the best fit to the data in this sense. Define $\chi^2$ ("chi-squared") as the sum of the squared differences between the $y_i$ data values and the $y$ values predicted by the model, as follows:

$$\chi^2 = \sum_{i=1}^{M} [y_i - (a + bx_i)]^2 \tag{C.11}$$

Then, the model parameters $a$ and $b$ defined by (C.9) and (C.10) yield the model function that minimizes $\chi^2$.

The linear regression process also computes $R$, the **correlation coefficient** between $x$ and $y$:

$$R = \frac{\sum_{i=1}^{M} (x_i - \overline{x})(y_i - \overline{y})}{\sqrt{\left(\sum_{i=1}^{M} (x_i - \overline{x})^2\right)\left(\sum_{i=1}^{M} (y_i - \overline{y})^2\right)}} \tag{C.12}$$

$R$ is a number between –1 and +1 that indicates how close to a straight line the $(x_i, y_i)$ data points fall. If the data points fall exactly on a line of positive slope (any slope), then $R$ is +1; we say $x$ and $y$ are **correlated**. If the data points fall exactly on a line of negative slope, then $R$ is –1; $x$ and $y$ are **anticorrelated**. If the data points do not fall on a straight line, then $R$'s value is somewhere between its extremes, and the farther the data points are from a straight line, the closer $R$ is to 0.

The correlation coefficient does not, however, tell whether the model (C.6) actually is a good fit to the data. $R$ only tells how closely $x$ and $y$ are correlated, *assuming* the model is a good fit. If a straight-line model is not a good fit—if, for example, the data looks more like a quadratic or an exponential function—then $R$ is meaningless. To decide whether a straight line is a good fit to the data, we can do a statistical goodness-of-fit test on the model. However, this test uses the probability distribution of the errors in the $y_i$ values, something we don't usually know. See any statistics textbook for further information about statistical goodness-of-fit tests.

Class edu.rit.numeric.XYSeries in the Parallel Java Library implements the linear regression formulas. Set up an instance of class XYSeries containing the $(x_i, y_i)$ data points, and then call the `linearRegression()` method to calculate $a$, $b$, and $R$.

## C.5 General Linear Least-Squares Curve Fitting

Generalizing the previous section, suppose we want to fit the data points $(x_i, y_i)$, $i$ = 1, 2, ..., $M$, to a model that is a linear combination of arbitrary functions of $x$:

$$y = a_1 f_1(x) + a_2 f_2(x) + \ldots + a_N f_N(x) \tag{C.13}$$

The functions $f_j(x)$, $j = 1, 2, ..., N$, are the model's **basis functions**, and the coefficients $a_j$, $j = 1, 2, ..., N$, are the **model parameters**. The linear regression model function is a special case of (C.13) with $N = 2$, $f_1(x) = 1$, and $f_2(x) = x$. In Chapter 10, the Second Problem Size Law was defined as the following:

$$T(N,K) = (a + bN) + \frac{1}{K}(c + dN) \tag{C.14}$$

This is a **multivariate model function** that again is a linear combination of basis functions

$$T(N,K) = a_1\, f_1(N,K) + a_2\, f_2(N,K) + a_3\, f_3(N,K) + a_4\, f_4(N,K) \tag{C.15}$$

where $f_1(N,K) = 1$, $f_2(N,K) = N$, $f_3(N,K) = 1/K$, and $f_4(N,K) = N/K$.

As we did for linear regression, define $\chi^2$ as the sum of the squared differences between the $y_i$ data values and the model $y$ values:

$$\chi^2 = \sum_{i=1}^{M}\left[ y_i - \left( \sum_{j=1}^{N} a_j f_j(x_i) \right) \right]^2 \tag{C.16}$$

Then, the **general linear least-squares** problem is to find the coefficients $a_j$ that minimize $\chi^2$.

To illustrate, suppose there are $M$=6 data points and $N$=3 basis functions. Writing Equation C.13 for each data point $(x_i, y_i)$ gives the following:

$$\begin{aligned}
a_1\, f_1(x_1) + a_2\, f_2(x_1) + a_3\, f_3(x_1) &= y_1 \\
a_1\, f_1(x_2) + a_2\, f_2(x_2) + a_3\, f_3(x_2) &= y_2 \\
a_1\, f_1(x_3) + a_2\, f_2(x_3) + a_3\, f_3(x_3) &= y_3 \\
a_1\, f_1(x_4) + a_2\, f_2(x_4) + a_3\, f_3(x_4) &= y_4 \\
a_1\, f_1(x_5) + a_2\, f_2(x_5) + a_3\, f_3(x_5) &= y_5 \\
a_1\, f_1(x_6) + a_2\, f_2(x_6) + a_3\, f_3(x_6) &= y_6
\end{aligned} \tag{C.17}$$

These are six simultaneous equations for the three unknowns $a_1$, $a_2$, and $a_3$. Because there are more equations than unknowns, it is, in general, not possible to find $a_1$, $a_2$, and $a_3$ that satisfy all the equations. The best we can do is to minimize the sum of the squared differences between the left sides and the right sides—that is, to minimize $\chi^2$.

The simultaneous equations can be written more compactly in matrix notation:

$$\mathbf{F} \cdot \mathbf{A} = \mathbf{Y} \tag{C.18}$$

The $M \times N$ matrix **F**, where each element $F_{ij} = f_j(x_i)$, is called the **design matrix** of the general linear least-squares problem. **A** is an $N$-element vector of the model parameters $a_j$. **Y** is an $M$-element vector of the $y_i$ data values. Given the data points and the basis functions, **F** and **Y** are known, and we are solving the matrix equation (C.18) for the unknown A.

In the general linear least-squares problems in this book, there are constraints on the model parameter values. Specifically, the model parameters must all be *nonnegative*. For example, it makes no sense for any of the Second Problem Size Law's model parameters to be negative. Thus, we want a solution to Equation C.18 that minimizes $\chi^2$, *and* such that A is nonnegative. If any model parameter $a_j$ "wants" to be negative, then that parameter is set to 0 instead, and the best solution for the remaining unconstrained parameters is found.

Charles Lawson and Richard Hanson have published a Non-Negative Least Squares algorithm that solves exactly this problem. They have also published a public domain Fortran program, `NNLS`, that implements the algorithm. Class edu.rit.numeric.NonNegativeLeastSquares in the Parallel Java Library is a translation of the `NNLS` program into Java. The TimeFit program mentioned in Chapter 10 uses class NonNegativeLeastSquares to fit a model for the running time $T(N,K)$ to a series of running-time measurements.

## C.6 Quadratic Equations

A **quadratic equation** is of the following form:

$$ax^2 + bx + c = 0 \tag{C.19}$$

We want to find the **roots** of the quadratic equation, that is, the value or values of $x$ that make the left side equal to 0. First, compute the **discriminant** of the quadratic equation, $D$:

$$D = b^2 - 4ac \tag{C.20}$$

If $D < 0$, then the quadratic equation has no (real) roots. If $D \geq 0$, then the quadratic equation has two roots $x_1$ and $x_2$, computed by the following formulas:

$$q = -\frac{1}{2}\left[b + \text{sgn}(b)\sqrt{D}\right] \tag{C.21}$$

$$x_1 = \frac{q}{a} \tag{C.22}$$

$$x_2 = \frac{c}{q} \tag{C.23}$$

sgn($b$) is the signum function; it is –1 if $b < 0$ and +1 if $b \geq 0$. The formulas (C.21)–(C.23) are less susceptible to roundoff error than the classical quadratic formula we all learned in high school.

Class edu.rit.numeric.Quadratic in the Parallel Java Library implements the preceding formulas. Use an instance of class Quadratic to find the roots of a quadratic equation.

## C.7 Cubic Equations

A **cubic equation** is of the following form:

$$dx^3 + ex^2 + fx + g = 0 \tag{C.24}$$

If $d$, the coefficient of $x^3$, is not 1, then divide both sides by $d$ to get

$$x^3 + ax^2 + bx + c = 0 \tag{C.25}$$

where $a = e/d$, $b = f/d$, and $c = g/d$. We want to find the roots of the cubic equation. First, compute two intermediate quantities $Q$ and $R$:

$$Q = \frac{3b - a^2}{9} \tag{C.26}$$

$$R = \frac{9ab - 27c - 2a^3}{54} \tag{C.27}$$

The discriminant of the cubic equation is the following:

$$D = Q^3 + R^2 \tag{C.28}$$

If $D < 0$, then the cubic equation has three unequal roots $x_1$, $x_2$, and $x_3$, computed by the following formulas:

$$\theta = \cos^{-1}\left(\frac{R}{\sqrt{-Q^3}}\right) \tag{C.29}$$

$$x_1 = 2\sqrt{-Q}\cos\left(\frac{\theta}{3}\right) - \frac{a}{3} \tag{C.30}$$

$$x_2 = 2\sqrt{-Q}\cos\left(\frac{\theta + 2\pi}{3}\right) - \frac{a}{3} \tag{C.31}$$

$$x_3 = 2\sqrt{-Q}\cos\left(\frac{\theta + 4\pi}{3}\right) - \frac{a}{3} \tag{C.32}$$

If $D = 0$, then the cubic equation has three roots $x_1$, $x_2$, and $x_3$, at least two of which are equal, computed by the following formulas:

$$x_1 = 2R^{1/3} - \frac{a}{3} \tag{C.33}$$

$$x_2 = x_3 = -R^{1/3} - \frac{a}{3} \tag{C.34}$$

If $D > 0$, then the cubic equation has one root $x_1$, computed by the following formulas:

$$S = \left(R + \sqrt{D}\right)^{1/3} \tag{C.35}$$

$$T = \left(R - \sqrt{D}\right)^{1/3} \tag{C.36}$$

$$x_1 = (S + T) - \frac{a}{3} \tag{C.37}$$

Class edu.rit.numeric.Cubic in the Parallel Java Library implements the preceding formulas. Use an instance of class Cubic to find the roots of a cubic equation.

## C.8 For Further Information

On numerical methods in general:

- W. Press, S. Teukolsky, W. Vetterling, and B. Flannery. *Numerical Recipes: The Art of Scientific Computing, Third Edition*. Cambridge University Press, 2008.

On libraries of numerical algorithms:

- GNU Scientific Library. http://www.gnu.org/software/gsl/

  (The GNU Scientific Library is free software released under the GNU General Public License.)
- Netlib Repository. http://www.netlib.org/

  (Some routines in the Netlib Repository are in the public domain, other routines are released under various software licenses.)
- Numerical Recipes Home Page. http://www.nr.com/

  (To use the *Numerical Recipes* routines, you must pay a license fee.)

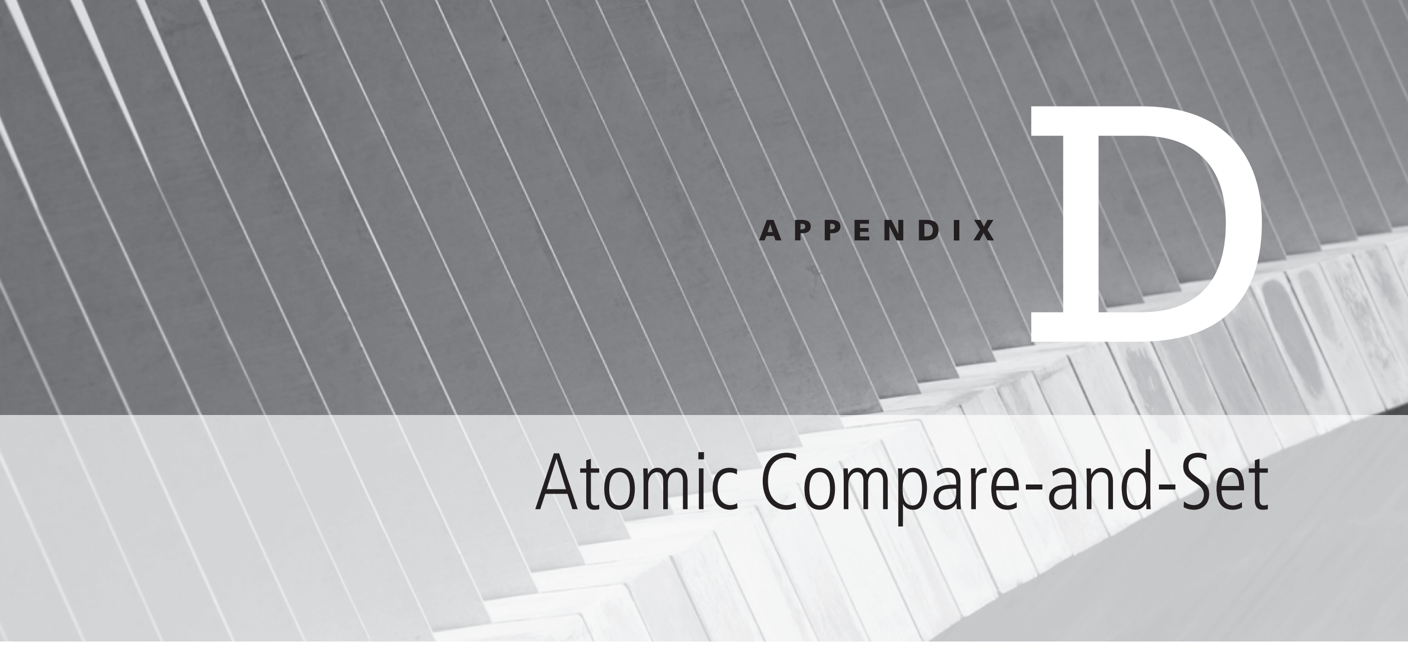
On the nonnegative least squares problem:

- C. Lawson and R. Hanson. *Solving Least Squares Problems*. Society for Industrial and Applied Mathematics, 1995.
- NNLS program. http://www.netlib.org/lawson-hanson/all

The formulas for the roots of quadratic and cubic equations come from:

- E. Weisstein. “Quadratic Equation.” From *MathWorld*—A Wolfram Web Resource. http://mathworld.wolfram.com/QuadraticEquation.html
- E. Weisstein. “Cubic formula.” From *MathWorld*—A Wolfram Web Resource. http://mathworld.wolfram.com/CubicFormula.html

*This page intentionally left blank*



APPENDIX D

# Atomic Compare-and-Set

in which we survey techniques for synchronizing multiple threads accessing a shared variable; we introduce the atomic compare-and-set operation; and we see how to update a shared variable in a multiple-thread-safe fashion using atomic compare-and-set

# APPENDIX D Atomic Compare-and-Set

## D.1 Blocking Synchronization

When multiple threads write a shared variable at the same time, or when some threads read the variable at the same time as other threads write the variable, the threads must synchronize with each other to avoid conflicts. One synchronization technique involves **blocking**. When thread A is about to access a shared variable, A is made to *block,* or suspend execution, if another thread B is already accessing the variable. When B finishes with the variable, A is allowed to *unblock,* or resume execution. A then proceeds to access the variable. In this way, only one thread at a time is permitted to access the variable, thus synchronizing the threads.

The Java platform provides several blocking constructs for synchronizing threads:

- **Synchronized methods**. Every Java object has an associated hidden lock. When a thread calls a `synchronized` method on an object, the thread blocks until the object's lock is unlocked, and then the thread locks the lock and executes the method (which contains statements to access the shared data, typically private fields of the object). When the thread returns from the method, the thread unlocks the lock.
- **Synchronized blocks**. A section of code can be placed inside a block that begins with the `synchronized` keyword designating an object.

```
synchronized (anObject)
   {
   // Statements to access shared data go here
   }
```

  When a thread arrives at a `synchronized` block, the thread blocks until the designated object's lock is unlocked, and then the thread locks the lock and executes the statements. When the thread finishes executing the statements, the thread unlocks the lock.

- **Semaphores**. Class java.util.concurrent.Semaphore provides a semaphore object. A semaphore is a counter. When a thread calls the semaphore's `acquire()` method, the `acquire()` method blocks the calling thread until the counter is greater than zero, and then the `acquire()` method decrements the counter and returns. At this point, the thread proceeds to access the shared data. When the thread finishes with the shared data, the thread calls the semaphore's `release()` method, which increments the counter.

- **Locks**. Class java.util.concurrent.locks.ReentrantLock provides a lock object. When a thread calls the lock's `lock()` method, the `lock()` method blocks the calling thread until the lock is unlocked, and then the `lock()` method locks the lock and returns. At this point, the thread proceeds to access the shared data. When the thread finishes with the shared data, the thread calls the lock's `unlock()` method, which unlocks the lock. (This is the same behavior as the `synchronized` keyword, except the lock is its own separate visible object.)

Other languages and operating systems provide similar blocking synchronization capabilities. Parallel Java programs can use the parallel region's `critical()` method for thread synchronization, as well as Java's built-in capabilities.

However, several problems arise when using locks, or any of the similar blocking techniques, to synchronize multiple threads in an SMP parallel program. These problems include the following:

- Locking and unlocking are heavyweight operations, requiring many CPU cycles to execute. This increases the program's running time—problematic in what is supposed to be a high-performance parallel program.
- Lock implementations are typically optimized for the no-contention case, where only one thread at a time ever tries to lock the lock. If several threads try to lock the lock, then the time needed to lock the lock goes up. This is a reasonable strategy if locking rarely occurs. But if locking is frequent, as is often the case in an SMP parallel program with multiple threads updating a shared variable, the lock implementation's reduced performance becomes problematic.
- Synchronization by blocking can lead to **convoying**. Suppose thread A locks a lock, and then a context switch happens and thread B starts executing. Suppose B tries to lock the lock and blocks because it is already locked, so a context switch happens and thread C starts executing. Suppose C tries to lock the lock . . . . Eventually a whole convoy of threads may pile up behind thread A. None of these threads can make any progress until A's turn to execute comes round again and A unlocks the lock. That's on a single-CPU machine. On an SMP parallel machine, each thread can have its own CPU, so the operating system does not have to context switch between different threads on the same CPU. However, depending on how the operating system's thread scheduler is implemented, convoying may still be possible to some extent even on an SMP machine.

## D.2 Atomic Compare-and-Set

An alternative, *non-blocking* thread synchronization technique uses the **atomic compare-and-set (CAS)** operation. CAS is performed on a given (shared) variable X, a given expected value, and a given updated value:

```
Compare-and-set (X, expected value, updated value)
    If X = expected value:
        X ← updated value
        Return true
    Else:
        Return false
```

A similar operation is **atomic compare-and-swap** or **atomic compare-and-exchange**:

```
Compare-and-swap (X, expected value, updated value)
    If X = expected value:
        Swap X ↔ updated value
        Return true
    Else:
        Return false
```

The preceding steps are *atomic;* they are guaranteed to execute with no interruptions or context switches in the middle. To ensure atomicity, CAS is usually a single CPU instruction executed by the hardware. Most CPUs, beginning with the IBM System/370 CPU in 1970, provide some kind of CAS instruction. The Intel 486 has the CMPXCHG instruction, which does an atomic compare-and-exchange on 32-bit quantities. The Intel Pentium added the CMPXCHG8B instruction, which does an atomic compare-and-exchange on 64-bit quantities. The Sun SPARC has the CASA and CASXA instructions.

The Java platform provides atomic compare-and-set operations on integers, longs, Booleans, and object references via the multiple-thread-safe wrapper classes in package java.util.concurrent.atomic. The Parallel Java Library provides atomic compare-and-set operations on all the primitive types, as well as object references, via the multiple-thread-safe wrapper classes in package edu.rit.pj.reduction. As an example, here are the definitions of two methods in class edu.rit.pj.reduction.SharedInteger.

```
/**
 * Returns this reduction variable's current value.
 *
 * @return  Current value.
 */
public int get();

/**
 * Atomically set this reduction variable to the given updated
 * value if the current value equals the expected value.
 *
 * @param  expect  Expected value.
 * @param  update  Updated value.
 *
 * @return  True if the update happened, false otherwise.
 */
public boolean compareAndSet
   (int expect,
    int update);
```

## D.3 Shared Variable Updating with Atomic CAS

Here's the pattern for using an atomic CAS operation to update a shared variable X in a multiple-thread-safe fashion:

```
1   Do:
2       Expected value ← Current value of X
3       Updated value ← New value of X
4   While CAS (X, expected value, updated value) is false
```

Note that there is no locking, so multiple threads can execute this sequence of statements simultaneously. The executing thread gets the current value of X, computes the new value of X (which may or may not depend on the current value), and does the atomic CAS operation. If no other thread changed the value of X between line 2 and line 4, X's value is still the same as the expected value, the CAS operation updates X to the new value and returns true, and the thread proceeds past the do-while loop. But if another thread changed the value of X between line 2 and line 4, the CAS operation leaves X unchanged and returns false, and the thread stays in the do-while loop. The thread rereads the current (changed) value of X, recomputes the new value of X (which may be different if X's value changed), and redoes the CAS operation. This loop continues until the CAS operation succeeds. A thread may have to spin through the retry loop several times if many threads are trying to update X, and the more threads there are, the more retries there may have to be.

As an example of the use of CAS to synchronize multiple threads updating a shared variable, here is the code that updates the bound variable in the SMP parallel branch-and-bound search program for maximum parsimony phylogenetic tree construction in Chapter 38. Afterward, bound is supposed to be set to the smaller of its current value and the value of `tentativeScore`.

```
// Atomically set global bound to the smaller of global bound
// and tentative score.
int newBound = bound.reduce
   (tentativeScore, IntegerOp.MINIMUM);
```

The `reduce()` method updates the shared variable in a multiple-thread-safe fashion. Rather than using blocking synchronization, the `reduce()` method uses a non-blocking CAS operation. Here is what reduce() does under the hood. `myValue` is a private field that holds the shared variable's value.

```
package edu.rit.pj.reduction;
import java.util.concurrent.atomic.AtomicInteger;
public class SharedInteger
   {
   private AtomicInteger myValue;

   public int reduce (int value, IntegerOp op)
      {
      for (;;)
         {
         int oldvalue = myValue.get();
         int newvalue = op.op (oldvalue, value);
         if (myValue.compareAndSet (oldvalue, newvalue))
            return newvalue;
         }
      }
   }
```

In the phylogenetic tree construction program, when the `reduce()` method calls the `op()` method on the `op` argument—that is, on the `IntegerOp.MINIMUM` object—the `op()` method returns the smaller of the current bound `(oldvalue)` and the tentative score `(value)`. The result becomes the updated value for the CAS operation. If the CAS operation succeeds, then the updated value is returned. If the CAS operation fails, then the loop repeats, the updated value is recalculated, and the CAS operation is retried.

## D.4 Limitations, Caveats

Because CAS is implemented in hardware and does not block the threads, a program that uses CAS for synchronization typically runs faster than a program that uses locking. On the other hand, locks can synchronize an arbitrary section of code, such as the code to add an item to a linked list; CAS can only synchronize an update of a single variable.

It is possible to write **lock-free concurrent data structures** using CAS as a building block. For example, a lock-free concurrent queue lets multiple threads add items to or remove items from the queue at the same time, without conflicts and without blocking. Some of the multiple thread safe collection classes in package java.util.concurrent use lock-free concurrent programming techniques. Space does not permit covering these techniques here. Herlihy's and Shavit's book is an excellent resource for further information.

A potential problem with using CAS to update a shared variable is the **ABA problem**. Here is the CAS updating pattern again:

```
Do:
    Expected value ← Current value of X
    Updated value ← New value of X
While CAS (X, expected value, updated value) is false
```

Suppose a thread executes line 2 and finds X's value to be A. The thread computes a new value for X at line 3. But before the thread can execute line 4, suppose another thread updates X's value to B. And further suppose that another thread updates X's value back to A. When the original thread executes line 4, X's value will be what the thread expects (A), so the CAS operation will succeed. However, this may not be correct, if the new value of X was computed assuming X would not be updated between line 2 and line 4.

The ABA problem does not occur if the sequence of values stored into X is *monotonic,* in the sense that once a certain value has been stored into X, that value will never be stored into X again. For example, if X is always incremented and never decremented, or if X is always set to the minimum of itself and another value, the sequence of X values will be monotonic and the ABA problem will not occur. The ABA problem can occur, for example, in linked-list data structures if the same nodes are linked, unlinked, and relinked into the list. Lock-free concurrent data structures must be implemented to avoid the ABA problem.

## D.5 For Further Information

A sampling of the vast literature on lock-free concurrent data structures:

- M. Michael and M. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing (PODC'96),* pages 267–275, 1996.
- M. Michael. High performance dynamic lock-free hash tables and list-based sets. *In Proceedings of the 14th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 73–82, 2002.
- D. Hendler, N. Shavit, and L. Yerushalmi. A scalable lock-free stack algorithm. In *Proceedings of the 16th Annual ACM Symposium on Parallelism in Algorithms and Architectures*, pages 206–215, 2004.
- M. Moir, D. Nussbaum, O. Shalev, and N. Shavit. Using elimination to implement scalable and lock-free FIFO queues. In *Proceedings of the 17th Annual ACM Symposium on Parallelism in Algorithms and Architectures*, pages 253–262, 2005.
- W. Scherer, D. Lea, and M. Scott. Scalable synchronous queues. In *Proceedings of the 11th Annual SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 147–156, 2006.

On concurrent programming on SMP machines:

- M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008.

*This page intentionally left blank*

# Index