Thomas Rauber
Gudula Rünger

Parallel Programming

for Multicore and Cluster Systems

Springer

Parallel Programming

Thomas Rauber · Gudula Rünger

Parallel Programming

For Multicore and Cluster Systems

Springer

Thomas Rauber
Universität Bayreuth
Computer Science Department
95440 Bayreuth
Germany
rauber@uni-bayreuth.de

Gudula Rünger
Technische Universität Chemnitz
Computer Science Department
09107 Chemnitz
Germany
ruenger@informatik.tu-chemnitz.de

ISBN 978-3-642-04817-3 e-ISBN 978-3-642-04818-0
DOI 10.1007/978-3-642-04818-0
Springer Heidelberg Dordrecht London New York

ACM Computing Classification (1998): D.1, C.1, C.2, C.4

Library of Congress Control Number: 2009941473

This is an extended English language translation of the German language edition:
Parallele Programmierung (2nd edn.) by T. Rauber and G. Rünger
Published in the book series: Springer-Lehrbuch

Cover design: KuenkelLopka GmbH, Heidelberg

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Preface

Innovations in hardware architecture, like hyperthreading or multicore processors, make parallel computing resources available for inexpensive desktop computers. However, the use of these innovations requires parallel programming techniques. In a few years, many standard software products will be based on concepts of parallel programming to use the hardware resources of future multicore processors efficiently. Thus, the need for parallel programming will extend to all areas of software development. The application area will be much larger than the area of scientific computing, which used to be the main area for parallel computing for many years. The expansion of the application area for parallel computing will lead to an enormous need for software developers with parallel programming skills. Some chip manufacturers already demand to include parallel programming as a standard course in computer science curricula.

This book takes up the new development in processor architecture by giving a detailed description of important parallel programming techniques that are necessary for developing efficient programs for multicore processors as well as for parallel cluster systems or supercomputers. Both shared and distributed address space architectures are covered. The main goal of the book is to present parallel programming techniques that can be used in many situations for many application areas and to enable the reader to develop correct and efficient parallel programs. Many example programs and exercises are provided to support this goal and to show how the techniques can be applied to further applications. The book can be used as both a textbook for students and a reference book for professionals. The material of the book has been used for courses in parallel programming at different universities for many years.

This is the third version of the book on parallel programming. The first two versions have been published in German in the years 2000 and 2007, respectively. This new English version is an updated and revised version of the newest German edition of the book. The update especially covers new developments in the area of multicore processors as well as a more detailed description of OpenMP and Java threads.

The content of the book consists of three main parts, covering all areas of parallel computing: the architecture of parallel systems, parallel programming models and environments, and the implementation of efficient application algorithms. The

emphasis lies on parallel programming techniques needed for different architectures.

The first part contains an overview of the architecture of parallel systems, including cache and memory organization, interconnection networks, routing and switching techniques, as well as technologies that are relevant for modern and future multicore processors.

The second part presents parallel programming models, performance models, and parallel programming environments for message passing and shared memory models, including MPI, Pthreads, Java threads, and OpenMP. For each of these parallel programming environments, the book gives basic concepts as well as more advanced programming methods and enables the reader to write and run semantically correct and efficient parallel programs. Parallel design patterns like pipelining, client–server, and task pools are presented for different environments to illustrate parallel programming techniques and to facilitate the implementation of efficient parallel programs for a wide variety of application areas. Performance models and techniques for runtime analysis are described in detail, as they are a prerequisite for achieving efficiency and high performance.

The third part applies the programming techniques from the second part to representative algorithms from scientific computing. The emphasis lies on basic methods for solving linear equation systems, which play an important role in many scientific simulations. The focus of the presentation lies on the analysis of the algorithmic structure of the different algorithms, which is the basis for parallelization, and not on the mathematical properties of the solution methods. For each algorithm, the book discusses different parallelization variants, using different methods and strategies.

Many colleagues and students have helped to improve the quality of this book. We would like to thank all of them for their help and constructive criticisms. For numerous corrections and suggestions we would like to thank Jörg Dümmler, Marvin Ferber, Michael Hofmann, Ralf Hoffmann, Sascha Hunold, Matthias Korch, Raphael Kunis, Jens Lang, John O'Donnell, Andreas Prell, Carsten Scholtes, and Michael Schwind. Many thanks to Matthias Korch, Carsten Scholtes, and Michael Schwind for help with the exercises. We thank Monika Glaser for her help and support with the LaTeX typesetting of the book. We also thank all the people who have been involved in the writing of the first two German versions of this book. It has been a pleasure working with the Springer Verlag in the development of this book. We especially thank Ralf Gerstner for his support and patience.

Bayreuth — Thomas Rauber
Chemnitz — Gudula Rünger
August 2009

Contents

Chapter 1
Introduction

In this short introduction, we give an overview of the use of parallelism and try to explain why parallel programming will be used for software development in the future. We also give an overview of the rest of the book and show how it can be used for courses with various foci.

1.1 Classical Use of Parallelism

Parallel programming and the design of efficient parallel programs have been well established in high-performance, scientific computing for many years. The simulation of scientific problems is an important area in natural and engineering sciences of growing importance. More precise simulations or the simulations of larger problems need greater and greater computing power and memory space. In the last decades, high-performance research included new developments in parallel hardware and software technologies, and a steady progress in parallel high-performance computing can be observed. Popular examples are simulations of weather forecast based on complex mathematical models involving partial differential equations or crash simulations from car industry based on finite element methods.

Other examples include drug design and computer graphics applications for film and advertising industry. Depending on the specific application, computer simulation is the main method to obtain the desired result or it is used to replace or enhance physical experiments. A typical example for the first application area is weather forecast where the future development in the atmosphere has to be predicted, which can only be obtained by simulations. In the second application area, computer simulations are used to obtain results that are more precise than results from practical experiments or that can be performed with less financial effort. An example is the use of simulations to determine the air resistance of vehicles: Compared to a classical wind tunnel experiment, a computer simulation can give more precise results because the relative movement of the vehicle in relation to the ground can be included in the simulation. This is not possible in the wind tunnel, since the vehicle cannot be moved. Crash tests of vehicles are an obvious example where computer simulations can be performed with less financial effort.

T. Rauber, G. Rünger, *Parallel Programming*,
DOI 10.1007/978-3-642-04818-0_1, © Springer-Verlag Berlin Heidelberg 2010

Computer simulations often require a large computational effort. A low performance of the computer system used can restrict the simulations and the accuracy of the results obtained significantly. In particular, using a high-performance system allows larger simulations which lead to better results. Therefore, parallel computers have often been used to perform computer simulations. Today, cluster systems built up from server nodes are widely available and are now often used for parallel simulations. To use parallel computers or cluster systems, the computations to be performed must be partitioned into several parts which are assigned to the parallel resources for execution. These computation parts should be independent of each other, and the algorithm performed must provide enough independent computations to be suitable for a parallel execution. This is normally the case for scientific simulations. To obtain a parallel program, the algorithm must be formulated in a suitable programming language. Parallel execution is often controlled by specific runtime libraries or compiler directives which are added to a standard programming language like C, Fortran, or Java. The programming techniques needed to obtain efficient parallel programs are described in this book. Popular runtime systems and environments are also presented.

1.2 Parallelism in Today's Hardware

Parallel programming is an important aspect of high-performance scientific computing but it used to be a niche within the entire field of hardware and software products. However, more recently parallel programming has left this niche and will become the mainstream of software development techniques due to a radical change in hardware technology.

Major chip manufacturers have started to produce processors with several power-efficient computing units on one chip, which have an independent control and can access the same memory concurrently. Normally, the term *core* is used for single computing units and the term *multicore* is used for the entire processor having several cores. Thus, using multicore processors makes each desktop computer a small parallel system. The technological development toward multicore processors was forced by physical reasons, since the clock speed of chips with more and more transistors cannot be increased at the previous rate without overheating.

Multicore architectures in the form of single multicore processors, shared memory systems of several multicore processors, or clusters of multicore processors with a hierarchical interconnection network will have a large impact on software development. In 2009, dual-core and quad-core processors are standard for normal desktop computers, and chip manufacturers have already announced the introduction of oct-core processors for 2010. It can be predicted from Moore's law that the number of cores per processor chip will double every 18–24 months. According to a report of Intel, in 2015 a typical processor chip will likely consist of dozens up to hundreds of cores where a part of the cores will be dedicated to specific purposes like network management, encryption and decryption, or graphics [109]; the

majority of the cores will be available for application programs, providing a huge performance potential.

The users of a computer system are interested in benefitting from the performance increase provided by multicore processors. If this can be achieved, they can expect their application programs to keep getting faster and keep getting more and more additional features that could not be integrated in previous versions of the software because they needed too much computing power. To ensure this, there should definitely be a support from the operating system, e.g., by using dedicated cores for their intended purpose or by running multiple user programs in parallel, if they are available. But when a large number of cores are provided, which will be the case in the near future, there is also the need to execute a single application program on multiple cores. The best situation for the software developer would be that there be an automatic transformer that takes a sequential program as input and generates a parallel program that runs efficiently on the new architectures. If such a transformer were available, software development could proceed as before. But unfortunately, the experience of the research in parallelizing compilers during the last 20 years has shown that for many sequential programs it is not possible to extract enough parallelism automatically. Therefore, there must be some help from the programmer, and application programs need to be restructured accordingly.

For the software developer, the new hardware development toward multicore architectures is a challenge, since existing software must be restructured toward parallel execution to take advantage of the additional computing resources. In particular, software developers can no longer expect that the increase of computing power can automatically be used by their software products. Instead, additional effort is required at the software level to take advantage of the increased computing power. If a software company is able to transform its software so that it runs efficiently on novel multicore architectures, it will likely have an advantage over its competitors.

There is much research going on in the area of parallel programming languages and environments with the goal of facilitating parallel programming by providing support at the right level of abstraction. But there are many effective techniques and environments already available. We give an overview in this book and present important programming techniques, enabling the reader to develop efficient parallel programs. There are several aspects that must be considered when developing a parallel program, no matter which specific environment or system is used. We give a short overview in the following section.

1.3 Basic Concepts

A first step in parallel programming is the design of a parallel algorithm or program for a given application problem. The design starts with the decomposition of the computations of an application into several parts, called **tasks**, which can be computed in parallel on the cores or processors of the parallel hardware. The decomposition into tasks can be complicated and laborious, since there are usually

many different possibilities of decomposition for the same application algorithm. The size of tasks (e.g., in terms of the number of instructions) is called **granularity** and there is typically the possibility of choosing tasks of different sizes. Defining the tasks of an application appropriately is one of the main intellectual works in the development of a parallel program and is difficult to automate. **Potential parallelism** is an inherent property of an application algorithm and influences how an application can be split into tasks.

The tasks of an application are coded in a parallel programming language or environment and are assigned to **processes** or **threads** which are then assigned to physical computation units for execution. The assignment of tasks to processes or threads is called **scheduling** and fixes the order in which the tasks are executed. Scheduling can be done by hand in the source code or by the programming environment, at compile time or dynamically at runtime. The assignment of processes or threads onto the physical units, processors or cores, is called **mapping** and is usually done by the runtime system but can sometimes be influenced by the programmer. The tasks of an application algorithm can be independent but can also depend on each other resulting in data or control dependencies of tasks. Data and control dependencies may require a specific execution order of the parallel tasks: If a task needs data produced by another task, the execution of the first task can start only after the other task has actually produced these data and has provided the information. Thus, dependencies between tasks are constraints for the scheduling. In addition, parallel programs need **synchronization** and coordination of threads and processes in order to execute correctly. The methods of synchronization and coordination in parallel computing are strongly connected with the way in which information is exchanged between processes or threads, and this depends on the memory organization of the hardware.

A coarse classification of the memory organization distinguishes between **shared memory** machines and **distributed memory** machines. Often the term *thread* is connected with shared memory and the term *process* is connected with distributed memory. For shared memory machines, a global shared memory stores the data of an application and can be accessed by all processors or cores of the hardware systems. Information exchange between threads is done by shared variables written by one thread and read by another thread. The correct behavior of the entire program has to be achieved by synchronization between threads so that the access to shared data is coordinated, i.e., a thread reads a data element not before the write operation by another thread storing the data element has been finalized. Depending on the programming language or environment, synchronization is done by the runtime system or by the programmer. For distributed memory machines, there exists a private memory for each processor, which can only be accessed by this processor, and no synchronization for memory access is needed. Information exchange is done by sending data from one processor to another processor via an interconnection network by explicit **communication operations**.

Specific **barrier operations** offer another form of coordination which is available for both shared memory and distributed memory machines. All processes or threads have to wait at a barrier synchronization point until all other processes or

threads have also reached that point. Only after all processes or threads have executed the code before the barrier, they can continue their work with the subsequent code after the barrier.

An important aspect of parallel computing is the **parallel execution time** which consists of the time for the computation on processors or cores and the time for data exchange or synchronization. The parallel execution time should be smaller than the sequential execution time on one processor so that designing a parallel program is worth the effort. The parallel execution time is the time elapsed between the start of the application on the first processor and the end of the execution of the application on all processors. This time is influenced by the distribution of work to processors or cores, the time for information exchange or synchronization, and **idle times** in which a processor cannot do anything useful but wait for an event to happen. In general, a smaller parallel execution time results when the work load is assigned equally to processors or cores, which is called **load balancing**, and when the overhead for information exchange, synchronization, and idle times is small. Finding a specific scheduling and mapping strategy which leads to a good load balance and a small overhead is often difficult because of many interactions. For example, reducing the overhead for information exchange may lead to load imbalance whereas a good load balance may require more overhead for information exchange or synchronization.

For a quantitative evaluation of the execution time of parallel programs, cost measures like **speedup** and **efficiency** are used, which compare the resulting parallel execution time with the sequential execution time on one processor. There are different ways to measure the cost or runtime of a parallel program and a large variety of parallel cost models based on parallel programming models have been proposed and used. These models are meant to bridge the gap between specific parallel hardware and more abstract parallel programming languages and environments.

1.4 Overview of the Book

The rest of the book is structured as follows. Chapter 2 gives an overview of important aspects of the hardware of parallel computer systems and addresses new developments like the trends toward multicore architectures. In particular, the chapter covers important aspects of memory organization with shared and distributed address spaces as well as popular interconnection networks with their topological properties. Since memory hierarchies with several levels of caches may have an important influence on the performance of (parallel) computer systems, they are covered in this chapter. The architecture of multicore processors is also described in detail. The main purpose of the chapter is to give a solid overview of the important aspects of parallel computer architectures that play a role in parallel programming and the development of efficient parallel programs.

Chapter 3 considers popular parallel programming models and paradigms and discusses how the inherent parallelism of algorithms can be presented to a parallel runtime environment to enable an efficient parallel execution. An important part of this chapter is the description of mechanisms for the coordination

of parallel programs, including synchronization and communication operations. Moreover, mechanisms for exchanging information and data between computing resources for different memory models are described. Chapter 4 is devoted to the performance analysis of parallel programs. It introduces popular performance or cost measures that are also used for sequential programs, as well as performance measures that have been developed for parallel programs. Especially, popular communication patterns for distributed address space architectures are considered and their efficient implementations for specific interconnection networks are given.

Chapter 5 considers the development of parallel programs for distributed address spaces. In particular, a detailed description of MPI (Message Passing Interface) is given, which is by far the most popular programming environment for distributed address spaces. The chapter describes important features and library functions of MPI and shows which programming techniques must be used to obtain efficient MPI programs. Chapter 6 considers the development of parallel programs for shared address spaces. Popular programming environments are Pthreads, Java threads, and OpenMP. The chapter describes all three and considers programming techniques to obtain efficient parallel programs. Many examples help to understand the relevant concepts and to avoid common programming errors that may lead to low performance or cause problems like deadlocks or race conditions. Programming examples and parallel programming pattern are presented. Chapter 7 considers algorithms from numerical analysis as representative example and shows how the sequential algorithms can be transferred into parallel programs in a systematic way.

The main emphasis of the book is to provide the reader with the programming techniques that are needed for developing efficient parallel programs for different architectures and to give enough examples to enable the reader to use these techniques for programs from other application areas. In particular, reading and using the book is a good training for software development for modern parallel architectures, including multicore architectures.

The content of the book can be used for courses in the area of parallel computing with different emphasis. All chapters are written in a self-contained way so that chapters of the book can be used in isolation; cross-references are given when material from other chapters might be useful. Thus, different courses in the area of parallel computing can be assembled from chapters of the book in a modular way. Exercises are provided for each chapter separately. For a course on the programming of multicore systems, Chaps. 2, 3, and 6 should be covered. In particular, Chapter 6 provides an overview of the relevant programming environments and techniques. For a general course on parallel programming, Chaps. 2, 5, and 6 can be used. These chapters introduce programming techniques for both distributed and shared address spaces. For a course on parallel numerical algorithms, mainly Chaps. 5 and 7 are suitable; Chap. 6 can be used additionally. These chapters consider the parallel algorithms used as well as the programming techniques required. For a general course on parallel computing, Chaps. 2, 3, 4, 5, and 6 can be used with selected applications from Chap. 7. The following web page will be maintained for additional and new material: `ai2.inf.uni-bayreuth.de/pp_book`.

Chapter 2
Parallel Computer Architecture

The possibility for parallel execution of computations strongly depends on the architecture of the execution platform. This chapter gives an overview of the general structure of parallel computers which determines how computations of a program can be mapped to the available resources such that a parallel execution is obtained. Section 2.1 gives a short overview of the use of parallelism within a single processor or processor core. Using the available resources within a single processor core at instruction level can lead to a significant performance increase. Sections 2.2 and 2.3 describe the control and data organization of parallel platforms. Based on this, Sect. 2.4.2 presents an overview of the architecture of multicore processors and describes the use of thread-based parallelism for simultaneous multithreading.

The following sections are devoted to specific components of parallel platforms. Section 2.5 describes important aspects of interconnection networks which are used to connect the resources of parallel platforms and to exchange data and information between these resources. Interconnection networks also play an important role in multicore processors for the connection between the cores of a processor chip. Section 2.5 describes static and dynamic interconnection networks and discusses important characteristics like diameter, bisection bandwidth, and connectivity of different network types as well as the embedding of networks into other networks. Section 2.6 addresses routing techniques for selecting paths through networks and switching techniques for message forwarding over a given path. Section 2.7 considers memory hierarchies of sequential and parallel platforms and discusses cache coherence and memory consistency for shared memory platforms.

2.1 Processor Architecture and Technology Trends

Processor chips are the key components of computers. Considering the trends observed for processor chips during the last years, estimations for future developments can be deduced. Internally, processor chips consist of transistors. The number of transistors contained in a processor chip can be used as a rough estimate of

T. Rauber, G. Rünger, *Parallel Programming*,
DOI 10.1007/978-3-642-04818-0_2, © Springer-Verlag Berlin Heidelberg 2010

its complexity and performance. **Moore's law** is an empirical observation which states that the number of transistors of a typical processor chip doubles every 18–24 months. This observation was first made by Gordon Moore in 1965 and is valid now for more than 40 years. The increasing number of transistors can be used for architectural improvements like additional functional units, more and larger caches, and more registers. A typical processor chip for desktop computers from 2009 consists of 400–800 million transistors.

The increase in the number of transistors has been accompanied by an increase in clock speed for quite a long time. Increasing the clock speed leads to a faster computational speed of the processor, and often the clock speed has been used as the main characteristic of the performance of a computer system. In the past, the increase in clock speed and in the number of transistors has led to an average performance increase of processors of 55% (integer operations) and 75% (floating-point operations), respectively [84]. This can be measured by specific benchmark programs that have been selected from different application areas to get a representative performance measure of computer systems. Often, the SPEC benchmarks (*System Performance and Evaluation Cooperative*) are used to measure the integer and floating-point performance of computer systems [137, 84], see `www.spec.org`. The average performance increase of processors exceeds the increase in clock speed. This indicates that the increasing number of transistors has led to architectural improvements which reduce the average time for executing an instruction. In the following, we give a short overview of such architectural improvements. Four phases of microprocessor design trends can be observed [35] which are mainly driven by the internal use of parallelism:

1. **Parallelism at bit level**: Up to about 1986, the word size used by processors for operations increased stepwise from 4 bits to 32 bits. This trend has slowed down and ended with the adoption of 64-bit operations beginning in the 1990s. This development has been driven by demands for improved floating-point accuracy and a larger address space. The trend has stopped at a word size of 64 bits, since this gives sufficient accuracy for floating-point numbers and covers a sufficiently large address space of 2^{64} bytes.
2. **Parallelism by pipelining**: The idea of pipelining at instruction level is an overlapping of the execution of multiple instructions. The execution of each instruction is partitioned into several steps which are performed by dedicated hardware units (pipeline stages) one after another. A typical partitioning could result in the following steps:

   (a) *fetch*: fetch the next instruction to be executed from memory;
   (b) *decode*: decode the instruction fetched in step (a);
   (c) *execute*: load the operands specified and execute the instruction;
   (d) *write-back*: write the result into the target register.

   An instruction pipeline is like an assembly line in automobile industry. The advantage is that the different pipeline stages can operate in parallel, if there are no control or data dependencies between the instructions to be executed, see

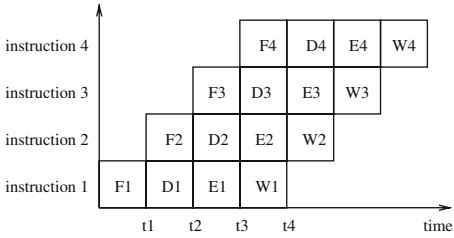


Fig. 2.1 Overlapping execution of four independent instructions by pipelining. The execution of each instruction is split into four stages: *fetch* (F), *decode* (D), *execute* (E), and *write-back* (W)

Fig. 2.1 for an illustration. To avoid waiting times, the execution of the different pipeline stages should take about the same amount of time. This time determines the cycle time of the processor. If there are no dependencies between the instructions, in each clock cycle the execution of one instruction is finished and the execution of another instruction started. The number of instructions finished per time unit is defined as the *throughput* of the pipeline. Thus, in the absence of dependencies, the throughput is one instruction per clock cycle.

In the absence of dependencies, all pipeline stages work in parallel. Thus, the number of pipeline stages determines the **degree of parallelism** attainable by a pipelined computation. The number of pipeline stages used in practice depends on the specific instruction and its potential to be partitioned into stages. Typical numbers of pipeline stages lie between 2 and 26 stages. Processors which use pipelining to execute instructions are called **ILP processors** (*instruction-level parallelism*). Processors with a relatively large number of pipeline stages are sometimes called **superpipelined**. Although the available degree of parallelism increases with the number of pipeline stages, this number cannot be arbitrarily increased, since it is not possible to partition the execution of the instruction into a very large number of steps of equal size. Moreover, data dependencies often inhibit a completely parallel use of the stages.

3. **Parallelism by multiple functional units**: Many processors are *multiple-issue processors*. They use multiple, independent functional units like ALUs (*arithmetic logical units*), FPUs (*floating-point units*), load/store units, or branch units. These units can work in parallel, i.e., different independent instructions can be executed in parallel by different functional units. Thus, the average execution rate of instructions can be increased. Multiple-issue processors can be distinguished into **superscalar** processors and **VLIW** (*very long instruction word*) processors, see [84, 35] for a more detailed treatment.

   The number of functional units that can efficiently be utilized is restricted because of data dependencies between neighboring instructions. For superscalar processors, these dependencies are determined at runtime dynamically by the hardware, and decoded instructions are dispatched to the instruction units using dynamic scheduling by the hardware. This may increase the complexity of the circuit significantly. Moreover, simulations have shown that superscalar processors with up to four functional units yield a substantial benefit over a single

functional unit. But using even more functional units provides little additional gain [35, 99] because of dependencies between instructions and branching of control flow.

4. **Parallelism at process or thread level**: The three techniques described so far assume a *single sequential* control flow which is provided by the compiler and which determines the execution order if there are dependencies between instructions. For the programmer, this has the advantage that a sequential programming language can be used nevertheless leading to a parallel execution of instructions. However, the degree of parallelism obtained by pipelining and multiple functional units is limited. This limit has already been reached for some time for typical processors. But more and more transistors are available per processor chip according to Moore's law. This can be used to integrate larger caches on the chip. But the cache sizes cannot be arbitrarily increased either, as larger caches lead to a larger access time, see Sect. 2.7.

   An alternative approach to use the increasing number of transistors on a chip is to put multiple, independent processor cores onto a single processor chip. This approach has been used for typical desktop processors since 2005. The resulting processor chips are called **multicore processors**. Each of the cores of a multicore processor must obtain a separate flow of control, i.e., parallel programming techniques must be used. The cores of a processor chip access the same memory and may even share caches. Therefore, memory accesses of the cores must be coordinated. The coordination and synchronization techniques required are described in later chapters.

A more detailed description of parallelism by multiple functional units can be found in [35, 84, 137, 164]. Section 2.4.2 describes techniques like simultaneous multithreading and multicore processors requiring an explicit specification of parallelism.

2.2 Flynn's Taxonomy of Parallel Architectures

Parallel computers have been used for many years, and many different architectural alternatives have been proposed and used. In general, a parallel computer can be characterized as a collection of processing elements that can communicate and cooperate to solve large problems fast [14]. This definition is intentionally quite vague to capture a large variety of parallel platforms. Many important details are not addressed by the definition, including the number and complexity of the processing elements, the structure of the interconnection network between the processing elements, the coordination of the work between the processing elements, as well as important characteristics of the problem to be solved.

For a more detailed investigation, it is useful to make a classification according to important characteristics of a parallel computer. A simple model for such a classification is given by **Flynn's taxonomy** [52]. This taxonomy characterizes parallel computers according to the global control and the resulting data and control flows. Four categories are distinguished:

1. **Single-Instruction, Single-Data (SISD)**: There is one processing element which has access to a single program and data storage. In each step, the processing element loads an instruction and the corresponding data and executes the instruction. The result is stored back in the data storage. Thus, SISD is the conventional sequential computer according to the *von Neumann model*.
2. **Multiple-Instruction, Single-Data (MISD)**: There are multiple processing elements each of which has a private program memory, but there is only one common access to a single global data memory. In each step, each processing element obtains the *same* data element from the data memory and loads an instruction from its private program memory. These possibly different instructions are then executed in parallel by the processing elements using the previously obtained (identical) data element as operand. This execution model is very restrictive and no commercial parallel computer of this type has ever been built.
3. **Single-Instruction, Multiple-Data (SIMD)**: There are multiple processing elements each of which has a private access to a (shared or distributed) data memory, see Sect. 2.3 for a discussion of shared and distributed address spaces. But there is only one program memory from which a special control processor fetches and dispatches instructions. In each step, each processing element obtains from the control processor the *same* instruction and loads a separate data element through its private data access on which the instruction is performed. Thus, the instruction is synchronously applied in parallel by all processing elements to different data elements.

   For applications with a significant degree of data parallelism, the SIMD approach can be very efficient. Examples are multimedia applications or computer graphics algorithms to generate realistic three-dimensional views of computer-generated environments.
4. **Multiple-Instruction, Multiple-Data (MIMD)**: There are multiple processing elements each of which has a separate instruction and data access to a (shared or distributed) program and data memory. In each step, each processing element loads a separate instruction and a separate data element, applies the instruction to the data element, and stores a possible result back into the data storage. The processing elements work asynchronously with each other. Multicore processors or cluster systems are examples for the MIMD model.

Compared to MIMD computers, SIMD computers have the advantage that they are easy to program, since there is only one program flow, and the synchronous execution does not require synchronization at program level. But the synchronous execution is also a restriction, since conditional statements of the form

```
if (b==0) c=a; else c = a/b;
```

must be executed in two steps. In the first step, all processing elements whose local value of `b` is zero execute the `then` part. In the second step, all other processing elements execute the `else` part. MIMD computers are more flexible, as each processing element can execute its own program flow. Most parallel computers

are based on the MIMD concept. Although Flynn's taxonomy only provides a coarse classification, it is useful to give an overview of the design space of parallel computers.

2.3 Memory Organization of Parallel Computers

Nearly all general-purpose parallel computers are based on the MIMD model. A further classification of MIMD computers can be done according to their memory organization. Two aspects can be distinguished: the physical memory organization and the view of the programmer of the memory. For the physical organization, computers with a physically shared memory (also called *multiprocessors*) and computers with a physically distributed memory (also called *multicomputers*) can be distinguished, see Fig. 2.2 for an illustration. But there also exist many hybrid organizations, for example providing a virtually shared memory on top of a physically distributed memory.

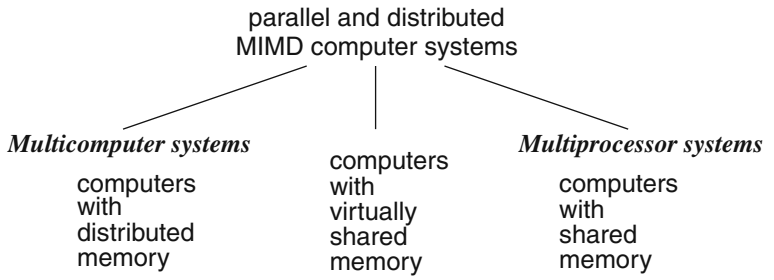


Fig. 2.2 Forms of memory organization of MIMD computers

From the programmer's point of view, it can be distinguished between computers with a distributed address space and computers with a shared address space. This view does not necessarily need to conform with the physical memory. For example, a parallel computer with a physically distributed memory may appear to the programmer as a computer with a shared address space when a corresponding programming environment is used. In the following, we have a closer look at the physical organization of the memory.

2.3.1 Computers with Distributed Memory Organization

Computers with a physically distributed memory are also called **distributed memory machines** (DMM). They consist of a number of processing elements (called nodes) and an interconnection network which connects nodes and supports the transfer of data between nodes. A node is an independent unit, consisting of processor, local memory, and, sometimes, periphery elements, see Fig. 2.3 (a) for an illustration.

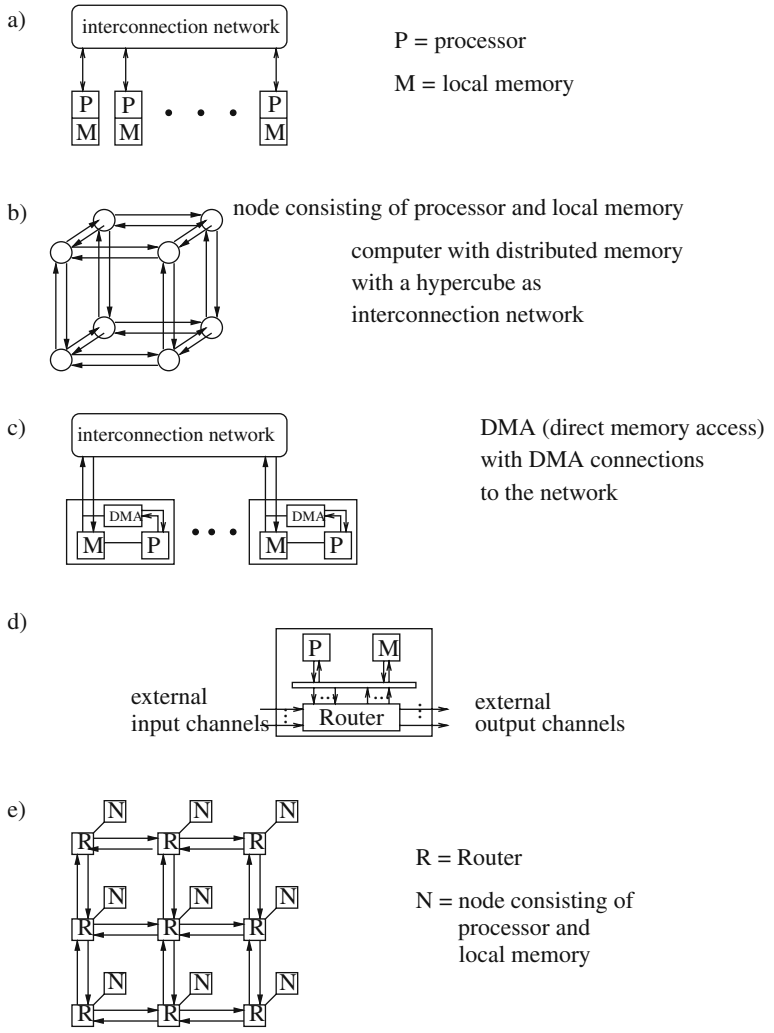


Fig. 2.3 Illustration of computers with distributed memory: (**a**) abstract structure, (**b**) computer with distributed memory and hypercube as interconnection structure, (**c**) DMA (direct memory access), (**d**) processor–memory node with router, and (**e**) interconnection network in the form of a mesh to connect the routers of the different processor–memory nodes

Program data is stored in the local memory of one or several nodes. All local memory is *private* and only the local processor can access the local memory directly. When a processor needs data from the local memory of other nodes to perform local computations, message-passing has to be performed via the interconnection network. Therefore, distributed memory machines are strongly connected with the message-passing programming model which is based on communication between cooperating sequential processes and which will be considered in more detail in

Chaps. 3 and 5. To perform message-passing, two processes P_A and P_B on different nodes A and B issue corresponding send and receive operations. When P_B needs data from the local memory of node A, P_A performs a send operation containing the data for the destination process P_B. P_B performs a receive operation specifying a receive buffer to store the data from the source process P_A from which the data is expected.

The architecture of computers with a distributed memory has experienced many changes over the years, especially concerning the interconnection network and the coupling of network and nodes. The interconnection network of earlier multicomputers were often based on **point-to-point connections** between nodes. A node is connected to a fixed set of other nodes by physical connections. The structure of the interconnection network can be represented as a graph structure. The nodes represent the processors, the edges represent the physical interconnections (also called *links*). Typically, the graph exhibits a regular structure. A typical network structure is the *hypercube* which is used in Fig. 2.3(b) to illustrate the node connections; a detailed description of interconnection structures is given in Sect. 2.5. In networks with point-to-point connection, the structure of the network determines the possible communications, since each node can only exchange data with its direct neighbor. To decouple send and receive operations, buffers can be used to store a message until the communication partner is ready. Point-to-point connections restrict parallel programming, since the network topology determines the possibilities for data exchange, and parallel algorithms have to be formulated such that their communication fits the given network structure [8, 115].

The execution of communication operations can be decoupled from the processor's operations by adding a **DMA controller** (DMA – direct memory access) to the nodes to control the data transfer between the local memory and the I/O controller. This enables data transfer from or to the local memory without participation of the processor (see Fig. 2.3(c) for an illustration) and allows asynchronous communication. A processor can issue a send operation to the DMA controller and can then continue local operations while the DMA controller executes the send operation. Messages are received at the destination node by its DMA controller which copies the enclosed data to a specific system location in local memory. When the processor then performs a receive operation, the data are copied from the system location to the specified receive buffer. Communication is still restricted to neighboring nodes in the network. Communication between nodes that do not have a direct connection must be controlled by software to send a message along a path of direct interconnections. Therefore, communication times between nodes that are not directly connected can be much larger than communication times between direct neighbors. Thus, it is still more efficient to use algorithms with communication according to the given network structure.

A further decoupling can be obtained by putting routers into the network, see Fig. 2.3(d). The routers form the actual network over which communication can be performed. The nodes are connected to the routers, see Fig. 2.3(e). Hardware-supported routing reduces communication times as messages for processors on remote nodes can be forwarded by the routers along a preselected path without

interaction of the processors in the nodes along the path. With router support, there is not a large difference in communication time between neighboring nodes and remote nodes, depending on the switching technique, see Sect. 2.6.3. Each physical I/O channel of a router can be used by one message only at a specific point in time. To decouple message forwarding, message buffers are used for each I/O channel to store messages and apply specific routing algorithms to avoid deadlocks, see also Sect. 2.6.1.

Technically, DMMs are quite easy to assemble since standard desktop computers can be used as nodes. The programming of DMMs requires a careful data layout, since each processor can directly access only its local data. Non-local data must be accessed via message-passing, and the execution of the corresponding send and receive operations takes significantly longer than a local memory access. Depending on the interconnection network and the communication library used, the difference can be more than a factor of 100. Therefore, data layout may have a significant influence on the resulting parallel runtime of a program. Data layout should be selected such that the number of message transfers and the size of the data blocks exchanged are minimized.

The structure of DMMs has many similarities with networks of workstations (NOWs) in which standard workstations are connected by a fast local area network (LAN). An important difference is that interconnection networks of DMMs are typically more specialized and provide larger bandwidths and lower latencies, thus leading to a faster message exchange.

Collections of complete computers with a dedicated interconnection network are often called **clusters**. Clusters are usually based on standard computers and even standard network topologies. The entire cluster is addressed and programmed as a single unit. The popularity of clusters as parallel machines comes from the availability of standard high-speed interconnections like FCS (Fiber Channel Standard), SCI (Scalable Coherent Interface), Switched Gigabit Ethernet, Myrinet, or InfiniBand, see [140, 84, 137]. A natural programming model of DMMs is the message-passing model that is supported by communication libraries like MPI or PVM, see Chap. 5 for a detailed treatment of MPI. These libraries are often based on standard protocols like TCP/IP [110, 139].

The difference between cluster systems and **distributed systems** lies in the fact that the nodes in cluster systems use the same operating system and can usually not be addressed individually; instead a special job scheduler must be used. Several cluster systems can be connected to **grid systems** by using middleware software like the Globus Toolkit, see `www.globus.org` [59]. This allows a coordinated collaboration of several clusters. In grid systems, the execution of application programs is controlled by the middleware software.

2.3.2 Computers with Shared Memory Organization

Computers with a physically shared memory are also called shared memory machines (SMMs); the shared memory is also called **global memory**. SMMs consist

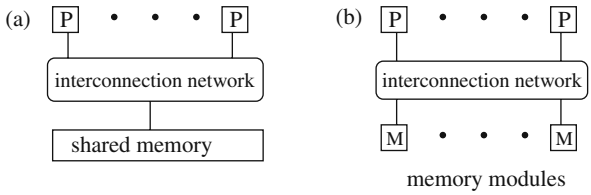


Fig. 2.4 Illustration of a computer with shared memory: (**a**) abstract view and (**b**) implementation of the shared memory with memory modules

of a number of processors or cores, a shared physical memory (global memory), and an interconnection network to connect the processors with the memory. The shared memory can be implemented as a set of memory modules. Data can be exchanged between processors via the global memory by reading or writing shared variables. The cores of a multicore processor are an example for an SMM, see Sect. 2.4.2 for a more detailed description. Physically, the global memory usually consists of separate memory modules providing a common address space which can be accessed by all processors, see Fig. 2.4 for an illustration.

A natural programming model for SMMs is the use of **shared variables** which can be accessed by all processors. Communication and cooperation between the processors is organized by writing and reading shared variables that are stored in the global memory. Accessing shared variables concurrently by several processors should be avoided since **race conditions** with unpredictable effects can occur, see also Chaps. 3 and 6.

The existence of a global memory is a significant advantage, since communication via shared variables is easy and since no data replication is necessary as is sometimes the case for DMMs. But technically, the realization of SMMs requires a larger effort, in particular because the interconnection network must provide fast access to the global memory for each processor. This can be ensured for a small number of processors, but scaling beyond a few dozen processors is difficult.

A special variant of SMMs are symmetric multiprocessors (SMPs). SMPs have a single shared memory which provides a uniform access time from any processor for all memory locations, i.e., all memory locations are equidistant to all processors [35, 84]. SMPs usually have a small number of processors that are connected via a central bus which also provides access to the shared memory. There are usually no private memories of processors or specific I/O processors, but each processor has a private cache hierarchy. As usual, access to a local cache is faster than access to the global memory. In the spirit of the definition from above, each multicore processor with several cores is an SMP system.

SMPs usually have only a small number of processors, since the central bus provides a constant bandwidth which is shared by all processors. When too many processors are connected, more and more access collisions may occur, thus increasing the effective memory access time. This can be alleviated by the use of caches and suitable cache coherence protocols, see Sect. 2.7.3. The maximum number of processors used in bus-based SMPs typically lies between 32 and 64.

Parallel programs for SMMs are often based on the execution of threads. A thread is a separate control flow which shares data with other threads via a global address

space. It can be distinguished between **kernel threads** that are managed by the operating system and **user threads** that are explicitly generated and controlled by the parallel program, see Sect. 3.7.2. The kernel threads are mapped by the operating system to processors for execution. User threads are managed by the specific programming environment used and are mapped to kernel threads for execution. The mapping algorithms as well as the exact number of processors can be hidden from the user by the operating system. The processors are completely controlled by the operating system. The operating system can also start multiple sequential programs from several users on different processors, when no parallel program is available. Small-size SMP systems are often used as servers, because of their cost-effectiveness, see [35, 140] for a detailed description.

SMP systems can be used as nodes of a larger parallel computer by employing an interconnection network for data exchange between processors of different SMP nodes. For such systems, a shared address space can be defined by using a suitable cache coherence protocol, see Sect. 2.7.3. A coherence protocol provides the view of a shared address space, although the physical memory might be distributed. Such a protocol must ensure that any memory access returns the most recently written value for a specific memory address, no matter where this value is physically stored. The resulting systems are also called distributed shared memory (DSM) architectures. In contrast to single SMP systems, the access time in DSM systems depends on the location of a data value in the global memory, since an access to a data value in the local SMP memory is faster than an access to a data value in the memory of another SMP node via the coherence protocol. These systems are therefore also called NUMAs (non-uniform memory access), see Fig. 2.5. Since single SMP systems have a uniform memory latency for all processors, they are also called UMAs (uniform memory access).

2.3.3 Reducing Memory Access Times

Memory access time has a large influence on program performance. This can also be observed for computer systems with a shared address space. Technological development with a steady reduction in the VLSI (very large scale integration) feature size has led to significant improvements in processor performance. Since 1980, integer performance on the SPEC benchmark suite has been increasing at about 55% per year, and floating-point performance at about 75% per year [84], see Sect. 2.1. Using the LINPACK benchmark, floating-point performance has been increasing at more than 80% per year. A significant contribution to these improvements comes from a reduction in processor cycle time. At the same time, the capacity of DRAM chips that are used for building main memory has been increasing by about 60% per year. In contrast, the access time of DRAM chips has only been decreasing by about 25% per year. Thus, memory access time does not keep pace with processor performance improvement, and there is an increasing gap between processor cycle time and memory access time. A suitable organization of memory access becomes

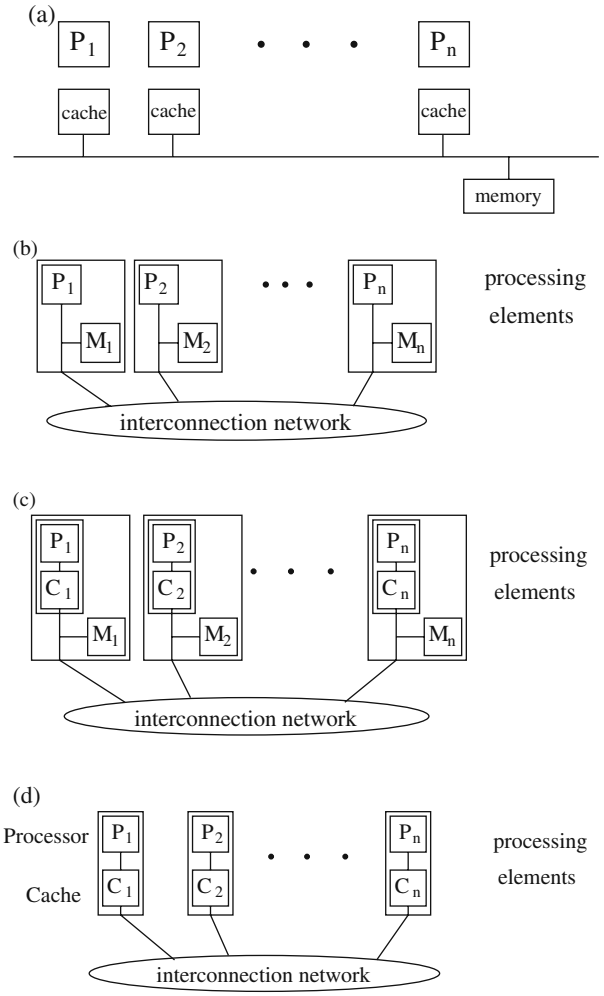


Fig. 2.5 Illustration of the architecture of computers with shared memory: (**a**) SMP – symmetric multiprocessors, (**b**) NUMA – non-uniform memory access, (**c**) CC-NUMA – cache-coherent NUMA, and (**d**) COMA – cache-only memory access

more and more important to get good performance results at program level. This is also true for parallel programs, in particular if a shared address space is used. Reducing the average latency observed by a processor when accessing memory can increase the resulting program performance significantly.

Two important approaches have been considered to reduce the average latency for memory access [14]: the simulation of **virtual processors** by each physical processor (multithreading) and the use of **local caches** to store data values that are accessed often. We give now a short overview of these approaches in the following.

2.3.3.1 Multithreading

The idea of **interleaved multithreading** is to hide the latency of memory accesses by simulating a fixed number of virtual processors for each physical processor. The physical processor contains a separate program counter (PC) as well as a separate set of registers for each virtual processor. After the execution of a machine instruction, an implicit switch to the next virtual processor is performed, i.e., the virtual processors are simulated by the physical processor in a round-robin fashion. The number of virtual processors per physical processor should be selected such that the time between the executions of successive instructions of a virtual processor is sufficiently large to load required data from the global memory. Thus, the memory latency will be hidden by executing instructions of other virtual processors. This approach does not reduce the amount of data loaded from the global memory via the network. Instead, instruction execution is organized such that a virtual processor accesses requested data not before their arrival. Therefore, from the point of view of a virtual processor, memory latency cannot be observed. This approach is also called **fine-grained multithreading**, since a switch is performed after each instruction. An alternative approach is **coarse-grained multithreading** which switches between virtual processors only on costly stalls, such as level 2 cache misses [84]. For the programming of fine-grained multithreading architectures, a PRAM-like programming model can be used, see Sect. 4.5.1. There are two drawbacks of fine-grained multithreading:

- The programming must be based on a large number of virtual processors. Therefore, the algorithm used must have a sufficiently large potential of parallelism to employ all virtual processors.
- The physical processors must be specially designed for the simulation of virtual processors. A software-based simulation using standard microprocessors is too slow.

There have been several examples for the use of fine-grained multithreading in the past, including Dencelor HEP (heterogeneous element processor) [161], NYU Ultracomputer [73], SB-PRAM [1], Tera MTA [35, 95], as well as the Sun T1 and T2 multiprocessors. For example, each T1 processor contains eight processor cores, each supporting four threads which act as virtual processors [84]. Section 2.4.1 will describe another variation of multithreading which is simultaneous multithreading.

2.3.3.2 Caches

A **cache** is a small, but fast memory between the processor and main memory. A cache can be used to store data that is often accessed by the processor, thus avoiding expensive main memory access. The data stored in a cache is always a subset of the data in the main memory, and the management of the data elements in the cache is done by hardware, e.g., by employing a set-associative strategy, see [84] and Sect. 2.7.1 for a detailed treatment. For each memory access issued by the processor, the hardware first checks whether the memory address specified currently resides

in the cache. If so, the data is loaded from the cache and no memory access is necessary. Therefore, memory accesses that go into the cache are significantly faster than memory accesses that require a load from the main memory. Since fast memory is expensive, several levels of caches are typically used, starting from a small, fast, and expensive level 1 (L1) cache over several stages (L2, L3) to the large, but slow main memory. For a typical processor architecture, access to the L1 cache only takes 2–4 cycles whereas access to main memory can take up to several hundred cycles. The primary goal of cache organization is to reduce the average memory access time as far as possible and to achieve an access time as close as possible to that of the L1 cache. Whether this can be achieved depends on the memory access behavior of the program considered, see Sect. 2.7.

Caches are used for single-processor computers, but they also play an important role in SMPs and parallel computers with different memory organization. SMPs provide a shared address space. If shared data is used by multiple processors, it may be replicated in multiple caches to reduce access latencies. Each processor should have a coherent view of the memory system, i.e., any read access should return the most recently written value no matter which processor has issued the corresponding write operation. A coherent view would be destroyed if a processor p changes the value of a memory address in its local cache without writing this value back to main memory. If another processor q would later read this memory address, it would not get the most recently written value. But even if p writes the value back to main memory, this may not be sufficient if q has a copy of the same memory location in its local cache. In this case, it is also necessary to update the copy in the local cache of q. The problem of providing a coherent view of the memory system is often referred to as **cache coherence problem**. To ensure cache coherency, a **cache coherency protocol** must be used, see Sect. 2.7.3 and [35, 84, 81] for a more detailed description.

2.4 Thread-Level Parallelism

The architectural organization within a processor chip may require the use of explicitly parallel programs to efficiently use the resources provided. This is called **thread-level parallelism**, since the multiple control flows needed are often called threads. The corresponding architectural organization is also called **chip multiprocessing** (CMP). An example for CMP is the placement of multiple independent **execution cores** with all execution resources onto a single processor chip. The resulting processors are called **multicore processors**, see Sect. 2.4.2.

An alternative approach is the use of *multithreading* to execute multiple threads simultaneously on a single processor by switching between the different threads when needed by the hardware. As described in Sect. 2.3.3, this can be obtained by fine-grained or coarse-grained multithreading. A variant of coarse-grained multithreading is **timeslice multithreading** in which the processor switches between the threads after a predefined timeslice interval has elapsed. This can lead to situations where the timeslices are not effectively used if a thread must wait for an event. If

this happens in the middle of a timeslice, the processor may remain unused for the rest of the timeslice because of the waiting. Such unnecessary waiting times can be avoided by using **switch-on-event multithreading** [119] in which the processor can switch to the next thread if the current thread must wait for an event to occur as can happen for cache misses.

A variant of this technique is **simultaneous multithreading** (SMT) which will be described in the following. This technique is called **hyperthreading** for some Intel processors. The technique is based on the observation that a single thread of control often does not provide enough instruction-level parallelism to use all functional units of modern superscalar processors.

2.4.1 Simultaneous Multithreading

The idea of simultaneous multithreading (SMT) is to use several threads and to schedule executable instructions from different threads in the same cycle if necessary, thus using the functional units of a processor more effectively. This leads to a simultaneous execution of several threads which gives the technique its name. In each cycle, instructions from several threads compete for the functional units of a processor. Hardware support for simultaneous multithreading is based on the replication of the chip area which is used to store the processor state. This includes the program counter (PC), user and control registers, as well as the interrupt controller with the corresponding registers. With this replication, the processor appears to the operating system and the user program as a set of **logical processors** to which processes or threads can be assigned for execution. These processes or threads can come from a single or several user programs. The number of replications of the processor state determines the number of logical processors.

Each logical processor stores its processor state in a separate processor resource. This avoids overhead for saving and restoring processor states when switching to another logical processor. All other resources of the processor chip like caches, bus system, and function and control units are shared by the logical processors. Therefore, the implementation of SMT only leads to a small increase in chip size. For two logical processors, the required increase in chip area for an Intel Xeon processor is less than 5% [119, 178]. The shared resources are assigned to the logical processors for simultaneous use, thus leading to a simultaneous execution of logical processors. When a logical processor must wait for an event, the resources can be assigned to another logical processor. This leads to a continuous use of the resources from the view of the physical processor. Waiting times for logical processors can occur for cache misses, wrong branch predictions, dependencies between instructions, and pipeline hazards.

Investigations have shown that the simultaneous use of processor resources by two logical processors can lead to performance improvements between 15% and 30%, depending on the application program [119]. Since the processor resources are shared by the logical processors, it cannot be expected that the use of more than two

logical processors can lead to a significant additional performance improvement. Therefore, SMT will likely be restricted to a small number of logical processors. Examples of processors that support SMT are the IBM Power5 and Power6 processors (two logical processors) and the Sun T1 and T2 processors (four/eight logical processors), see, e.g., [84] for a more detailed description.

To use SMT to obtain performance improvements, it is necessary that the operating system be able to control logical processors. From the point of view of the application program, it is necessary that every logical processor has a separate thread available for execution. Therefore, the application program must apply parallel programming techniques to get performance improvements for SMT processors.

2.4.2 Multicore Processors

According to Moore's law, the number of transistors of a processor chip doubles every 18–24 months. This enormous increase has enabled hardware manufacturers for many years to provide a significant performance increase for application programs, see also Sect. 2.1. Thus, a typical computer is considered old-fashioned and too slow after at most 5 years, and customers buy new computers quite often. Hardware manufacturers are therefore trying to keep the obtained performance increase at least at the current level to avoid reduction in computer sales figures.

As discussed in Sect. 2.1, the most important factors for the performance increase per year have been an increase in clock speed and the internal use of parallel processing like pipelined execution of instructions and the use of multiple functional units. But these traditional techniques have mainly reached their limits:

- Although it is possible to put additional functional units on the processor chip, this would not increase performance for most application programs because dependencies between instructions of a single control thread inhibit their parallel execution. A single control flow does not provide enough instruction-level parallelism to keep a large number of functional units busy.
- There are two main reasons why the speed of processor clocks cannot be increased significantly [106]. First, the increase in the number of transistors on a chip is mainly achieved by increasing the transistor density. But this also increases the power density and heat production because of leakage current and power consumption, thus requiring an increased effort and more energy for cooling. Second, memory access time could not be reduced at the same rate as the processor clock period. This leads to an increased number of machine cycles for a memory access. For example, in 1990 main memory access was between 6 and 8 cycles for a typical desktop computer system, whereas in 2006 memory access typically took between 100 and 250 cycles, depending on the DRAM technology used to build the main memory. Therefore, memory access times could become a limiting factor for further performance increase, and cache memories are used to prevent this, see Sect. 2.7 for a further discussion.

There are more problems that processor designers have to face: Using the increased number of transistors to increase the complexity of the processor architecture may also lead to an increase in processor–internal wire length to transfer control and data between the functional units of the processor. Here, the speed of signal transfers within the wires could become a limiting factor. For example, a 3 GHz processor has a cycle time of 0.33 ns. Assuming a signal transfer at the speed of light ($0.3 \cdot 10^9$ m/s), a signal can cross a distance of $0.33 \cdot 10^{-9}$ s $\cdot 0.3 \cdot 10^9$ m/s $=$ 10 cm in one processor cycle. This is not significantly larger than the typical size of a processor chip, and wire lengths become an important issue.

Another problem is the following: The physical size of a processor chip limits the number of pins that can be used, thus limiting the bandwidth between CPU and main memory. This may lead to a processor-to-memory performance gap which is sometimes referred to as *memory wall*. This makes the use of high-bandwidth memory architectures with an efficient cache hierarchy necessary [17].

All these reasons inhibit a processor performance increase at the previous rate using the traditional techniques. Instead, new processor architectures have to be used, and the use of multiple cores on a single processor die is considered as the most promising approach. Instead of further increasing the complexity of the internal organization of a processor chip, this approach integrates multiple independent processing cores with a relatively simple architecture onto one processor chip. This has the additional advantage that the energy consumption of a processor chip can be reduced if necessary by switching off unused processor cores during idle times [83].

Multicore processors integrate multiple execution cores on a single processor chip. For the operating system, each execution core represents an independent logical processor with separate execution resources like functional units or execution pipelines. Each core has to be controlled separately, and the operating system can assign different application programs to the different cores to obtain a parallel execution. Background applications like virus checking, image compression, and encoding can run in parallel to application programs of the user. By using techniques of parallel programming, it is also possible to execute a computation-intensive application program (like computer games, computer vision, or scientific simulations) in parallel on a set of cores, thus reducing the execution time compared to an execution on a single core or leading to more accurate results by performing more computations as in the sequential case. In the future, users of standard application programs as computer games will likely expect an efficient use of the execution cores of a processor chip. To achieve this, programmers have to use techniques from parallel programming.

The use of multiple cores on a single processor chip also enables standard programs, like text processing, office applications, or computer games, to provide additional features that are computed in the background on a separate core so that the user does not notice any delay in the main application. But again, techniques of parallel programming have to be used for the implementation.

2.4.3 Architecture of Multicore Processors

There are many different design variants for multicore processors, differing in the number of cores, the structure and size of the caches, the access of cores to caches, and the use of heterogeneous components. From a high-level view, three different types of architectures can be distinguished, and there are also hybrid organizations [107].

2.4.3.1 Hierarchical Design

For a hierarchical design, multiple cores share multiple caches. The caches are organized in a tree-like configuration, and the size of the caches increases from the leaves to the root, see Fig. 2.6 (left) for an illustration. The root represents the connection to external memory. Thus, each core can have a separate L1 cache and shares the L2 cache with other cores. All cores share the common external memory, resulting in a three-level hierarchy as illustrated in Fig. 2.6 (left). This can be extended to more levels. Additional sub-components can be used to connect the caches of one level with each other. A typical usage area for a hierarchical design is the SMP configuration.

A hierarchical design is also often used for standard desktop or server processors. Examples are the IBM Power6 architecture, the processors of the Intel Xeon and AMD Opteron family, as well as the Sun Niagara processors (T1 and T2). Figure 2.7 shows the design of the Quad-Core AMD Opteron and the Intel Quad-Core Xeon processors as a typical example for desktop processors with a hierarchical design. Many graphics processing units (GPUs) also exhibit a hierarchical design. An example is shown in Fig. 2.8 for the Nvidia GeForce 8800, which has 128 stream processors (SP) at 1.35 GHz organized in 8 texture/processor clusters (TPC) such that each TPC contains 16 SPs. This architecture is scalable to smaller and larger configurations by scaling the number of SPs and memory partitions, see [137] for a detailed description.

This figure will be printed in b/w

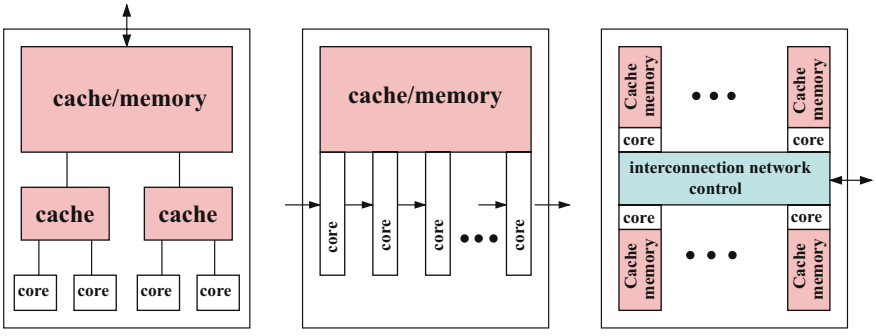
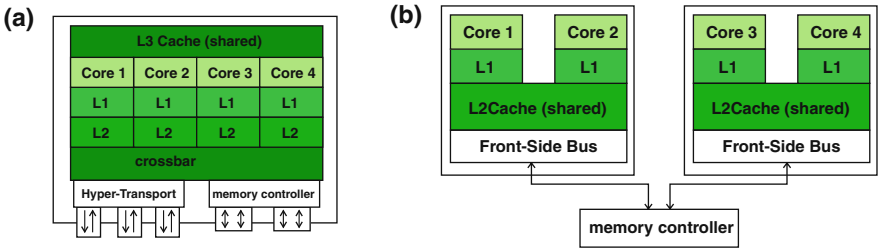
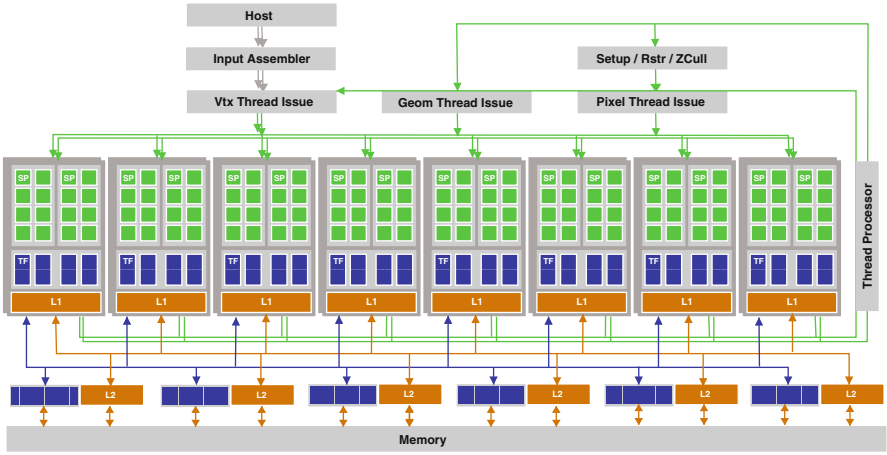


Fig. 2.6 Design choices for multicore chips according to [107]



This figure will be printed in b/w

Fig. 2.7 Quad-Core AMD Opteron (*left*) vs. Intel Quad-Core Xeon architecture (*right*) as examples for a hierarchical design



This figure will be printed in b/w

Fig. 2.8 Architectural overview of Nvidia GeForce 8800, see [128, 137] for a detailed description

2.4.3.2 Pipelined Designs

For a pipelined design, data elements are processed by multiple execution cores in a pipelined way. Data elements enter the processor chip via an input port and are passed successively through different cores until the processed data elements leave the last core and the entire processor chip via an output port, see Fig. 2.6 (middle). Each core performs specific processing steps on each data element.

Pipelined designs are useful for application areas in which the same computation steps have to be applied to a long sequence of data elements. Network processors used in routers and graphics processors both perform this style of computations. Examples for network processors with a pipelined design are the Xelerator X10 and X11 processors [176, 107] for the successive processing of network packets in a pipelined way within the chip. The Xelerator X11 contains up to 800 separate cores which are arranged in a logically linear pipeline, see Fig. 2.9 for an illustration. The network packets to be processed enter the chip via multiple input ports on one side of the chip, are successively processed by the cores, and then exit the chip.

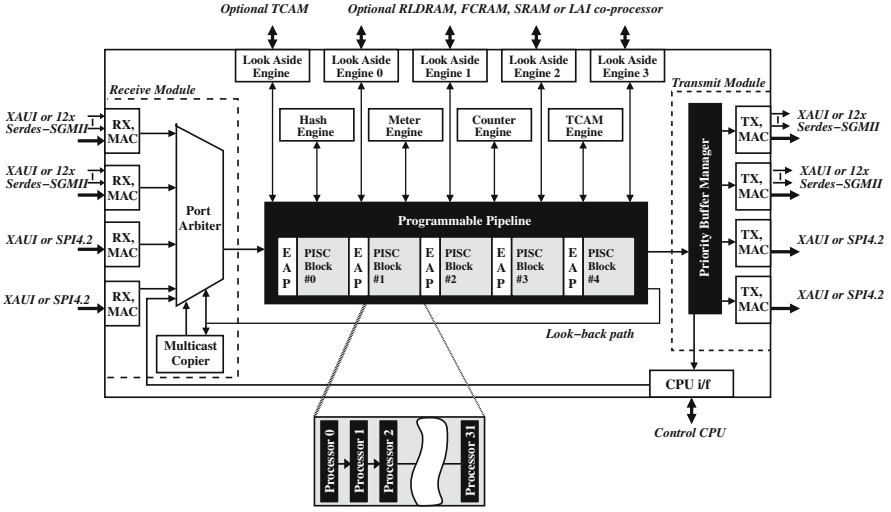


Fig. 2.9 Xelerator X11 network processor as an example for a pipelined design [176]

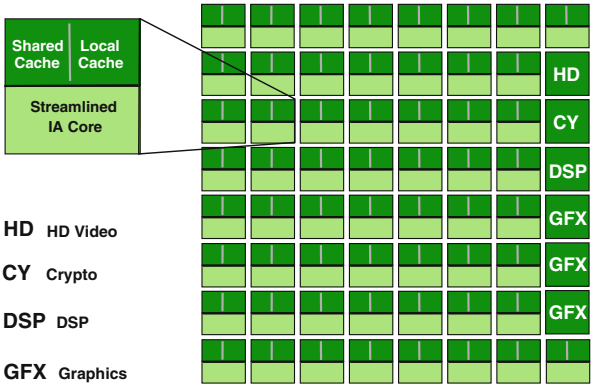
2.4.3.3 Network-Based Design

For a network-based design, the cores of a processor chip and their local caches and memories are connected via an interconnection network with other cores of the chip, see Fig. 2.6 (right) for an illustration. Data transfer between the cores is performed via the interconnection network. This network may also provide support for the synchronization of the cores. Off-chip interfaces may be provided via specialized cores or DMA ports. An example for a network-based design is the Intel Teraflop processor, which has been designed by the Intel Tera-scale Computing Research Program [83, 17].

This research program addresses the challenges of building processor chips with tens to hundreds of execution cores, including core design, energy management, cache and memory hierarchy, and I/O. The Teraflop processor developed as a prototype contains 80 cores, which are arranged in a 8×10 mesh, see Fig. 2.10 for an illustration. Each core can perform floating-point operations and contains a local cache as well as a router to perform data transfer between the cores and the main memory. There are additional cores for processing video data, encryption, and graphics computations. Depending on the application area, the number of specialized cores of such a processor chip could be varied.

2.4.3.4 Future Trends and Developments

The potential of multicore processors has been realized by most processor manufacturers like Intel or AMD, and since about 2005, many manufacturers deliver processors with two or more cores. Since 2007, Intel and AMD provide quad-core processors (like the Quad-Core AMD Opteron and the Quad-Core Intel Xeon), and



This figure will be printed in b/w

Fig. 2.10 Intel Teraflop processor according to [83] as an example for a network-based design of a multicore processor

the provision of oct-core processors is expected in 2010. The IBM Cell processor integrates one standard desktop core based on the Power Architecture and eight specialized processing cores. The UltraSPARC T2 processor from Sun has up to eight processing cores each of which can simulate eight threads using SMT (which is called CoolThreads by Sun). Thus, an UltraSPARC T2 processor can simultaneously execute up to 64 threads.

An important issue for the integration of a large number of cores in one processor chip is an efficient on-chip interconnection, which provides enough bandwidth for data transfers between the cores [83]. This interconnection should be *scalable* to support an increasing number of cores for future generations of processor designs and *robust* to tolerate failures of specific cores. If one or a few cores exhibit hardware failures, the rest of the cores should be able to continue operation. The interconnection should also support an efficient energy management which allows the scale-down of power consumption of individual cores by reducing the clock speed.

For an efficient use of processing cores, it is also important that the data to be processed be transferred to the cores fast enough to avoid the cores to wait for the data to be available. Therefore, an efficient memory system and I/O system are important. The memory system may use private first-level (L1) caches which can only be accessed by their associated cores, as well as shared second-level (L2) caches which can contain data of different cores. In addition, a shared third-level (L3) cache is often used. Processor chip with dozens or hundreds of cores will likely require an additional level of caches in the memory hierarchy to fulfill bandwidth requirements [83]. The I/O system must be able to provide enough bandwidth to keep all cores busy for typical application programs. At the physical layer, the I/O system must be able to bring hundreds of gigabits per second onto the chip. Such powerful I/O systems are currently under development [83].

Table 2.1 gives a short overview of typical multicore processors in 2009. For a more detailed treatment of the architecture of multicore processors and further examples, we refer to [137, 84].

Table 2.1 Examples for multicore processors in 2009

Processor	Number of cores	Number of threads	Clock GHz	L1 cache	L2 cache	L3 cache	Year released
Intel Xeon E5450 "Harpertown"	4	4	3.0	4× 32 KB	2× 6.1 MB		2007
Intel Xeon E5540 "Gainestown"	4	8	2.53	4× 64 KB	4× 256 MB	8 MB	2009
AMD Opteron "Barcelona"	4	4	2.0	4× 64 KB	4× 512 KB	2 MB	2007
AMD Opteron "Istanbul"	6	6	2.8	6× 128 KB	6× 512 KB	6 MB	2009
IBM Power6	2	4	4.7	128 KB	2× 4 MB	32 MB	2007
Sun T2 Niagara 2	8	64	1.17	8× 8 KB	4 MB		2007

2.5 Interconnection Networks

A physical connection between the different components of a parallel system is provided by an **interconnection network**. Similar to control flow and data flow, see Sect. 2.2, or memory organization, see Sect. 2.3, the interconnection network can also be used for a classification of parallel systems. Internally, the network consists of links and switches which are arranged and connected in some regular way. In multicomputer systems, the interconnection network is used to connect the processors or nodes with each other. Interactions between the processors for coordination, synchronization, or exchange of data are obtained by communication through message-passing over the links of the interconnection network. In multiprocessor systems, the interconnection network is used to connect the processors with the memory modules. Thus, memory accesses of the processors are performed via the interconnection network.

In both cases, the main task of the interconnection network is to transfer a message from a specific processor to a specific destination. The message may contain data or a memory request. The destination may be another processor or a memory module. The requirement for the interconnection network is to perform the message transfer correctly as fast as possible, even if several messages have to be transferred at the same time. Message transfer and memory accesses represent a significant part of operations of parallel systems with a distributed or shared address space. Therefore, the interconnection network used represents a significant part of the design of a parallel system and may have a large influence on its performance. Important design criteria of networks are

- the **topology** describing the interconnection structure used to connect different processors or processors and memory modules and
- the **routing technique** describing the exact message transmission used within the network between processors or processors and memory modules.

The topology of an interconnection network describes the geometric structure used for the arrangement of switches and links to connect processors or processors and memory modules. The geometric structure can be described as a graph in which switches, processors, or memory modules are represented as vertices and physical links are represented as edges. It can be distinguished between *static* and *dynamic* interconnection networks. **Static interconnection networks** connect nodes (processors or memory modules) *directly* with each other by fixed physical links. They are also called **direct networks** or **point-to-point networks**. The number of connections to or from a node may vary from only one in a star network to the total number of nodes in the network for a completely connected graph, see Sect. 2.5.2. Static networks are often used for systems with a distributed address space where a node comprises a processor and the corresponding memory module. **Dynamic interconnection networks** connect nodes *indirectly* via switches and links. They are also called **indirect networks**. Examples of indirect networks are *bus-based networks* or *switching networks* which consist of switches connected by links. Dynamic networks are used for both parallel systems with distributed and shared address space. Often, hybrid strategies are used [35].

The routing technique determines *how* and *along which path* messages are transferred in the network from a sender to a receiver. A path in the network is a series of nodes along which the message is transferred. Important aspects of the routing technique are the **routing algorithm** which determines the path to be used for the transmission and the **switching strategy** which determines whether and how messages are cut into pieces, how a routing path is assigned to a message, and how a message is forwarded along the processors or switches on the routing path.

The combination of routing algorithm, switching strategy, and network topology determines the performance of a network significantly. In Sects. 2.5.2 and 2.5.4, important direct and indirect networks are described in more detail. Specific routing algorithms and switching strategies are presented in Sects. 2.6.1 and 2.6.3. Efficient algorithms for the realization of common communication operations on different static networks are given in Chap. 4. A more detailed treatment of interconnection networks is given in [19, 35, 44, 75, 95, 115, 158].

2.5.1 Properties of Interconnection Networks

Static interconnection networks use fixed links between the nodes. They can be described by a connection graph $G = (V, E)$ where V is a set of nodes to be connected and E is a set of direct connection links between the nodes. If there is a direct physical connection in the network between the nodes $u \in V$ and $v \in V$, then it is $(u, v) \in E$. For most parallel systems, the interconnection network is *bidirectional*. This means that along a physical link messages can be transferred in both directions at the same time. Therefore, the connection graph is usually defined as an undirected graph. When a message must be transmitted from a node u to a node v and there is no direct connection between u and v in the network, a path from u to v must be selected which consists of several intermediate nodes along which the message

is transferred. A sequence of nodes $(v_0, \ldots, v_k)$ is called *path* of length k between v_0 and v_k, if $(v_i, v_{i+1}) \in E$ for $0 \leq i < k$. For parallel systems, all interconnection networks fulfill the property that there is at least one path between any pair of nodes $u, v \in V$.

Static networks can be characterized by specific properties of the connection graph, including the following properties: number of nodes, diameter of the network, degree of the nodes, bisection bandwidth, node and edge connectivity of the network, and flexibility of embeddings into other networks as well as the embedding of other networks. In the following, a precise definition of these properties is given.

The **diameter** $\delta(G)$ of a network G is defined as the maximum distance between any pair of nodes:

$$\delta(G) = \max_{u,v \in V} \min_{\substack{\varphi \text{ path} \\ \text{from } u \text{ to } v}} \{k \mid k \text{ is the length of the path } \varphi \text{ from } u \text{ to } v\}.$$

The diameter of a network determines the length of the paths to be used for message transmission between any pair of nodes. The **degree** $g(G)$ of a network G is the maximum degree of a node of the network where the degree of a node n is the number of direct neighbor nodes of n:

$$g(G) = \max\{g(v) \mid g(v) \text{ degree of } v \in V\}.$$

In the following, we assume that $|A|$ denotes the number of elements in a set A. The **bisection bandwidth** $B(G)$ of a network G is defined as the minimum number of edges that must be removed to partition the network into two parts of equal size without any connection between the two parts. For an uneven total number of nodes, the size of the parts may differ by 1. This leads to the following definition for $B(G)$:

$$B(G) = \min_{\substack{U_1, U_2 \text{ partition of } V \\ ||U_1| - |U_2|| \leq 1}} |\{(u, v) \in E \mid u \in U_1, v \in U_2\}|.$$

$B(G) + 1$ messages can saturate a network G, if these messages must be transferred at the same time over the corresponding edges. Thus, bisection bandwidth is a measure for the capacity of a network when transmitting messages simultaneously.

The **node** and **edge connectivity** of a network measure the number of nodes or edges that must fail to disconnect the network. A high connectivity value indicates a high reliability of the network and is therefore desirable. Formally, the node connectivity of a network is defined as the minimum number of nodes that must be deleted to disconnect the network, i.e., to obtain two unconnected network parts (which do not necessarily need to have the same size as is required for the bisection bandwidth). For an exact definition, let $G_{V \setminus M}$ be the rest graph which is obtained by deleting all nodes in $M \subset V$ as well as all edges adjacent to these nodes. Thus, it is $G_{V \setminus M} = (V \setminus M, E \cap ((V \setminus M) \times (V \setminus M)))$. The node connectivity $nc(G)$ of G is then defined as

$$nc(G) = \min_{M \subset V} \{|M| \mid \text{ there exist } u, v \in V \setminus M, \text{ such that there exists no path in } G_{V \setminus M} \text{ from } u \text{ to } v\}.$$

Similarly, the edge connectivity of a network is defined as the minimum number of edges that must be deleted to disconnect the network. For an arbitrary subset $F \subset E$, let $G_{E \setminus F}$ be the rest graph which is obtained by deleting the edges in F, i.e., it is $G_{E \setminus F} = (V, E \setminus F)$. The edge connectivity $ec(G)$ of G is then defined as

$$ec(G) = \min_{F \subset E} \{|F| \mid \text{ there exist } u, v \in V, \text{ such that there exists no path in } G_{E \setminus F} \text{ from } u \text{ to } v\}.$$

The node and edge connectivity of a network is a measure of the number of independent paths between any pair of nodes. A high connectivity of a network is important for its availability and reliability, since many nodes or edges can fail before the network is disconnected. The minimum degree of a node in the network is an upper bound on the node or edge connectivity, since such a node can be completely separated from its neighboring nodes by deleting all incoming edges. Figure 2.11 shows that the node connectivity of a network can be smaller than its edge connectivity.

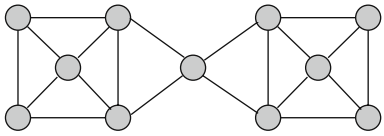


Fig. 2.11 Network with node connectivity 1, edge connectivity 2, and degree 4. The smallest degree of a node is 3

The flexibility of a network can be captured by the notion of **embedding**. Let $G = (V, E)$ and $G' = (V', E')$ be two networks. An embedding of G' into G assigns each node of G' to a node of G such that different nodes of G' are mapped to different nodes of G and such that edges between two nodes in G' are also present between their associated nodes in G [19]. An embedding of G' into G can formally be described by a mapping function $\sigma : V' \to V$ such that the following holds:

- if $u \neq v$ for $u, v \in V'$, then $\sigma(u) \neq \sigma(v)$ and
- if $(u, v) \in E'$, then $(\sigma(u), \sigma(v)) \in E$.

If a network G' can be embedded into a network G, this means that G is at least as flexible as G', since any algorithm that is based on the network structure of G', e.g., by using edges between nodes for communication, can be re-formulated for G with the mapping function σ, thus using corresponding edges in G for communication.

The network of a parallel system should be designed to meet the requirements formulated for the architecture of the parallel system based on typical usage patterns. Generally, the following topological properties are desirable:

- a small diameter to ensure small distances for message transmission,
- a small node degree to reduce the hardware overhead for the nodes,
- a large bisection bandwidth to obtain large data throughputs,

- a large connectivity to ensure reliability of the network,
- embedding into a large number of networks to ensure flexibility, and
- easy extendability to a larger number of nodes.

Some of these properties are conflicting and there is no network that meets all demands in an optimal way. In the following, some popular direct networks are presented and analyzed. The topologies are illustrated in Fig. 2.12. The topological properties are summarized in Table 2.2.

2.5.2 Direct Interconnection Networks

Direct interconnection networks usually have a regular structure which is transferred to their graph representation $G = (V, E)$. In the following, we use $n = |V|$ for the number of nodes in the network and use this as a parameter of the network type considered. Thus, each network type captures an entire class of networks instead of a fixed network with a given number of nodes.

A **complete graph** is a network G in which each node is directly connected with every other node, see Fig. 2.12(a). This results in diameter $\delta(G) = 1$ and degree $g(G) = n - 1$. The node and edge connectivity is $nc(G) = ec(G) = n - 1$, since a node can only be disconnected by deleting all $n - 1$ adjacent edges or neighboring nodes. For even values of n, the bisection bandwidth is $B(G) = n^2/4$: If two subsets of nodes of size $n/2$ each are built, there are $n/2$ edges from each of the nodes of one subset into the other subset, resulting in $n/2 \cdot n/2$ edges between the subsets. All other networks can be embedded into a complete graph, since there is a connection between any two nodes. Because of the large node degree, complete graph networks can only be built physically for a small number of nodes.

In a **linear array network**, nodes are arranged in a sequence and there is a bidirectional connection between any pair of neighboring nodes, see Fig. 2.12(b), i.e., it is $V = \{v_1, \ldots, v_n\}$ and $E = \{(v_i, v_{i+1}) \mid 1 \leq i < n\}$. Since $n - 1$ edges have to be traversed to reach v_n starting from v_1, the diameter is $\delta(G) = n - 1$. The connectivity is $nc(G) = ec(G) = 1$, since the elimination of one node or edge disconnects the network. The network degree is $g(G) = 2$ because of the inner nodes, and the bisection bandwidth is $B(G) = 1$. A linear array network can be embedded in nearly all standard networks except a tree network, see below. Since there is a link only between neighboring nodes, a linear array network does not provide fault tolerance for message transmission.

In a **ring network**, nodes are arranged in ring order. Compared to the linear array network, there is one additional bidirectional edge from the first node to the last node, see Fig. 2.12(c). The resulting diameter is $\delta(G) = \lfloor n/2 \rfloor$, the degree is $g(G) = 2$, the connectivity is $nc(G) = ec(G) = 2$, and the bisection bandwidth is also $B(G) = 2$. In practice, ring networks can be used for small number of processors and as part of more complex networks.

A **d-dimensional mesh** (also called **d-dimensional array**) for $d \geq 1$ consists of $n = n_1 \cdot n_2 \cdot \ldots \cdot n_d$ nodes that are arranged as a d-dimensional mesh, see

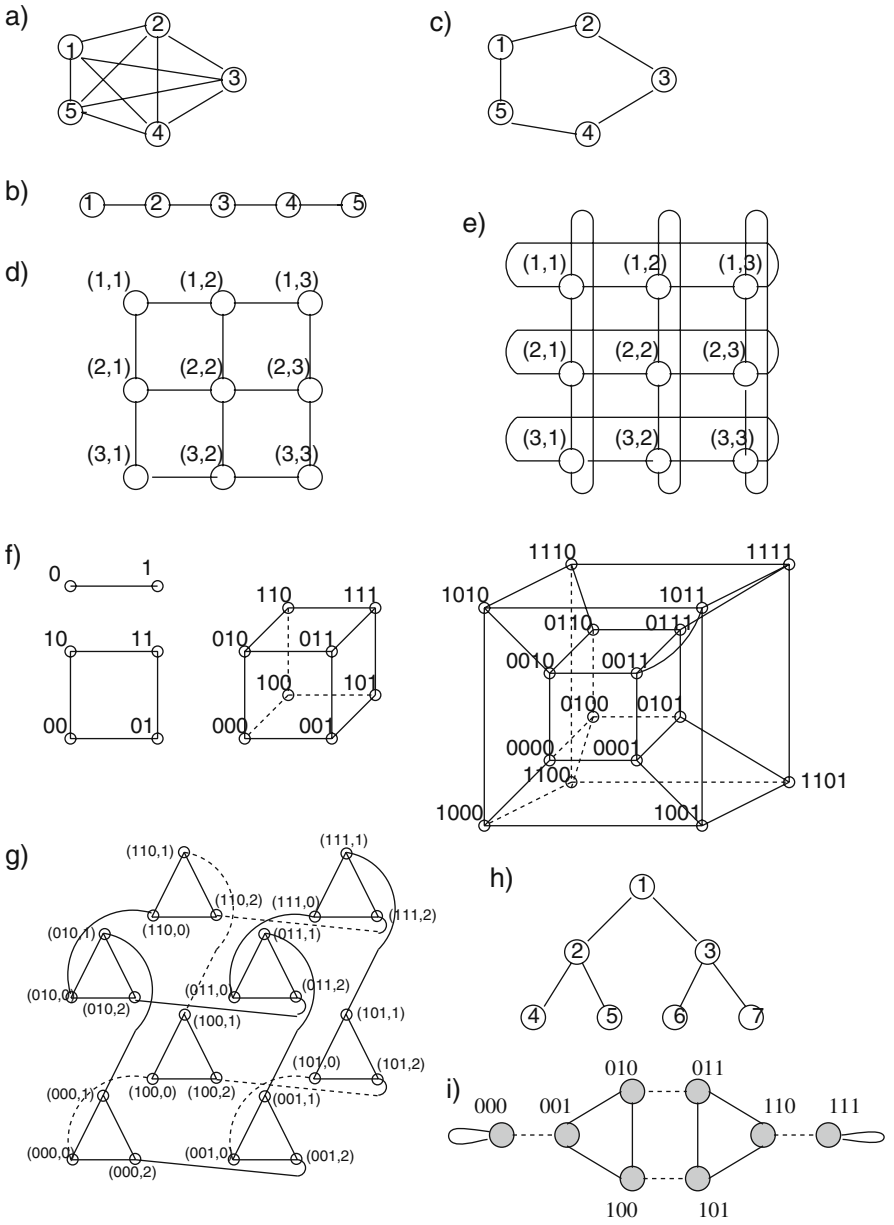


Fig. 2.12 Static interconnection networks: (**a**) complete graph, (**b**) linear array, (**c**) ring, (**d**) two-dimensional mesh, (**e**) two-dimensional torus, (**f**) k-dimensional cube for k=1,2,3,4, (**g**) cube-connected-cycles network for $k = 3$, (**h**) complete binary tree, (**i**) shuffle–exchange network with 8 nodes, where dashed edges represent exchange edges and straight edges represent shuffle edges

Table 2.2 Summary of important characteristics of static interconnection networks for selected topologies

Network G with n nodes	Degree $g(G)$	Diameter $\delta(G)$	Edge- connectivity $ec(G)$	Bisection bandwidth $B(G)$
Complete graph	$n-1$	1	$n-1$	$\left(\frac{n}{2}\right)^2$
Linear array	2	$n-1$	1	1
Ring	2	$\lfloor \frac{n}{2} \rfloor$	2	2
d-Dimensional mesh ($n = r^d$)	$2d$	$d(\sqrt[d]{n}-1)$	d	$n^{\frac{d-1}{d}}$
d-Dimensional torus ($n = r^d$)	$2d$	$d\left\lfloor \frac{\sqrt[d]{n}}{2} \right\rfloor$	$2d$	$2n^{\frac{d-1}{d}}$
k-Dimensional hypercube ($n = 2^k$)	$\log n$	$\log n$	$\log n$	$\frac{n}{2}$
k-Dimensional CCC network ($n = k2^k$ for $k \geq 3$)	3	$2k - 1 + \lfloor k/2 \rfloor$	3	$\frac{n}{2k}$
Complete binary tree ($n = 2^k - 1$)	3	$2\log \frac{n+1}{2}$	1	1
k-ary d-cube ($n = k^d$)	$2d$	$d\lfloor \frac{k}{2} \rfloor$	$2d$	$2k^{d-1}$

Fig. 2.12(d). The parameter n_j denotes the extension of the mesh in dimension j for $j = 1, \ldots, d$. Each node in the mesh is represented by its position $(x_1, \ldots, x_d)$ in the mesh with $1 \leq x_j \leq n_j$ for $j = 1, \ldots, d$. There is an edge between node $(x_1, \ldots, x_d)$ and $(x'_1, \ldots x'_d)$, if there exists $\mu \in \{1, \ldots, d\}$ with

$$|x_\mu - x'_\mu| = 1 \text{ and } x_j = x'_j \text{ for all } j \neq \mu.$$

In the case that the mesh has the same extension in all dimensions (also called *symmetric mesh*), i.e., $n_j = r = \sqrt[d]{n}$ for all $j = 1, \ldots, d$, and therefore $n = r^d$, the network diameter is $\delta(G) = d \cdot (\sqrt[d]{n} - 1)$, resulting from the path length between nodes on opposite sides of the mesh. The node and edge connectivity is $nc(G) = ec(G) = d$, since the corner nodes of the mesh can be disconnected by deleting all d incoming edges or neighboring nodes. The network degree is $g(G) = 2d$, resulting from inner mesh nodes which have two neighbors in each dimension. A two-dimensional mesh has been used for the Teraflop processor from Intel, see Sect. 2.4.3.

A d-**dimensional torus** is a variation of a d-dimensional mesh. The difference is the additional edges between the first and the last node in each dimension, i.e., for each dimension $j = 1, \ldots, d$ there is an edge between node $(x_1, \ldots, x_{j-1}, 1, x_{j+1}, \ldots, x_d)$ and $(x_1, \ldots, x_{j-1}, n_j, x_{j+1}, \ldots, x_d)$, see Fig. 2.12(e). For the symmetric case $n_j = \sqrt[d]{n}$ for all $j = 1, \ldots, d$, the diameter of the torus network is $\delta(G) = d \cdot \lfloor \sqrt[d]{n}/2 \rfloor$. The node degree is $2d$ for each node, i.e., $g(G) = 2d$. Therefore, node and edge connectivities are also $nc(G) = ec(G) = 2d$.

A k-**dimensional cube** or **hypercube** consists of $n = 2^k$ nodes which are connected by edges according to a recursive construction, see Fig. 2.12(f). Each

node is represented by a binary word of length k, corresponding to the numbers $0, \ldots, 2^k - 1$. A one-dimensional cube consists of two nodes with bit representations 0 and 1 which are connected by an edge. A k-dimensional cube is constructed from two given $(k-1)$-dimensional cubes, each using binary node representations $0, \ldots, 2^{k-1} - 1$. A k-dimensional cube results by adding edges between each pair of nodes with the same binary representation in the two $(k-1)$-dimensional cubes. The binary representations of the nodes in the resulting k-dimensional cube are obtained by adding a leading 0 to the previous representation of the first $(k-1)$-dimensional cube and adding a leading 1 to the previous representations of the second $(k-1)$-dimensional cube. Using the binary representations of the nodes $V = \{0,1\}^k$, the recursive construction just mentioned implies that there is an edge between node $\alpha_0 \ldots \alpha_j \ldots \alpha_{k-1}$ and node $\alpha_0 \ldots \bar{\alpha_j} \ldots \alpha_{k-1}$ for $0 \leq j \leq k-1$ where $\bar{\alpha_j} = 1$ for $\alpha_j = 0$ and $\bar{\alpha_j} = 0$ for $\alpha_j = 1$. Thus, there is an edge between every pair of nodes whose binary representation differs in exactly one bit position. This fact can also be captured by the Hamming distance.

The **Hamming distance** of two binary words of the same length is defined as the number of bit positions in which their binary representations differ. Thus, two nodes of a k-dimensional cube are directly connected, if their Hamming distance is 1. Between two nodes $v, w \in V$ with Hamming distance d, $1 \leq d \leq k$, there exists a path of length d connecting v and w. This path can be determined by traversing the bit representation of v bitwise from left to right and inverting the bits in which v and w differ. Each bit inversion corresponds to a traversal of the corresponding edge to a neighboring node. Since the bit representation of any two nodes can differ in at most k positions, there is a path of length $\leq k$ between any pair of nodes. Thus, the diameter of a k-dimensional cube is $\delta(G) = k$. The node degree is $g(G) = k$, since a binary representation of length k allows k bit inversions, i.e., each node has exactly k neighbors. The node and edge connectivity is $nc(G) = ec(G) = k$ as will be described in the following.

The connectivity of a hypercube is at most k, i.e., $nc(G) \leq k$, since each node can be completely disconnected from its neighbors by deleting all k neighbors or all k adjacent edges. To show that the connectivity is at least k, we show that there are exactly k independent paths between any pair of nodes v and w. Two paths are independent of each other if they do not share any edge, i.e., independent paths between v and w only share the two nodes v and w. The independent paths are constructed based on the binary representations of v and w, which are denoted by A and B, respectively, in the following. We assume that A and B differ in l positions, $1 \leq l \leq k$, and that these are the first l positions (which can be obtained by a renumbering). We can construct l paths of length l each between v and w by inverting the first l bits of A in different orders. For path i, $0 \leq i < l$, we stepwise invert bits $i, \ldots, l-1$ in this order first, and then invert bits $0, \ldots, i-1$ in this order. This results in l independent paths. Additional $k-l$ independent paths between v and w of length $l+2$ each can be constructed as follows: For i with $0 \leq i < k-l$, we first invert the bit $(l+i)$ of A and then the bits at positions $0, \ldots, l-1$ stepwise. Finally, we invert the bit $(l+i)$ again, obtaining bit representation B. This is shown

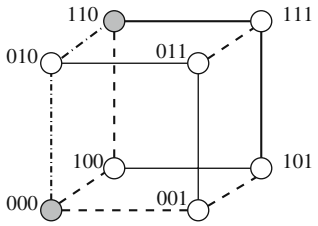


Fig. 2.13 In a three-dimensional cube network, we can construct three independent paths (from node 000 to node 110). The Hamming distance between node 000 and node 110 is $l = 2$. There are two independent paths between 000 and 110 of length $l = 2$: path (000, 100, 110) and path (000, 010, 110). Additionally, there are $k - l = 1$ path of length $l + 2 = 4$: path (000, 001, 101, 111, 110)

in Fig. 2.13 for an example. All k paths constructed are independent of each other, showing that $nc(G) \geq k$ holds.

A k-dimensional cube allows the embedding of many other networks as will be shown in the next subsection.

A **cube-connected cycles** (CCC) network results from a k-dimensional cube by replacing each node with a cycle of k nodes. Each of the nodes in the cycle has one off-cycle connection to one neighbor of the original node of the k-dimensional cube, thus covering all neighbors, see Fig. 2.12(g). The nodes of a CCC network can be represented by $V = \{0, 1\}^k \times \{0, \ldots, k-1\}$ where $\{0, 1\}^k$ are the binary representations of the k-dimensional cube and $i \in \{0, \ldots, k-1\}$ represents the position in the cycle. It can be distinguished between cycle edges F and cube edges E:

$$F = \{((\alpha, i), (\alpha, (i+1) \bmod k)) \mid \alpha \in \{0, 1\}^k, 0 \leq i < k\},$$
$$E = \{((\alpha, i), (\beta, i)) \mid \alpha_i \neq \beta_i \text{ and } \alpha_j = \beta_j \text{ for } j \neq i\}.$$

Each of the $k \cdot 2^k$ nodes of the CCC network has degree $g(G) = 3$, thus eliminating a drawback of the k-dimensional cube. The connectivity is $nc(G) = ec(G) = 3$ since each node can be disconnected by deleting its three neighboring nodes or edges. An upper bound for the diameter is $\delta(G) = 2k - 1 + \lfloor k/2 \rfloor$. To construct a path of this length, we consider two nodes in two different cycles with maximum hypercube distance k. These are nodes (α, i) and (β, j) for which α and β differ in all k bits. We construct a path from (α, i) to (β, j) by sequentially traversing a cube edge and a cycle edge for each bit position. The path starts with $(\alpha_0 \ldots \alpha_i \ldots \alpha_{k-1}, i)$ and reaches the next node by inverting α_i to $\bar{\alpha_i} = \beta_i$. From $(\alpha_0 \ldots \beta_i \ldots \alpha_{k-1}, i)$ the next node $(\alpha_0 \ldots \beta_i \ldots \alpha_{k-1}, (i+1) \bmod k)$ is reached by using a cycle edge. In the next steps, the bits $\alpha_{i+1}, \ldots, \alpha_{k-1}$ and $\alpha_0, \ldots, \alpha_{i-1}$ are successively inverted in this way, using a cycle edge between the steps. This results in $2k - 1$ edge traversals. Using at most $\lfloor k/2 \rfloor$ additional traversals of cycle edges starting from $(\beta, i + k - 1 \bmod k)$ leads to the target node (β, j).

A **complete binary tree** network has $n = 2^k - 1$ nodes which are arranged as a binary tree in which all leaf nodes have the same depth, see Fig. 2.12(h). The

degree of inner nodes is 3, leading to a total degree of $g(G) = 3$. The diameter of the network is $\delta(G) = 2 \cdot \log \frac{n+1}{2}$ and is determined by the path length between two leaf nodes in different subtrees of the root node; the path consists of a subpath from the first leaf to the root followed by a subpath from the root to the second leaf. The connectivity of the network is $nc(G) = ec(G) = 1$, since the network can be disconnected by deleting the root or one of the edges to the root.

A k-dimensional **shuffle–exchange** network has $n = 2^k$ nodes and $3 \cdot 2^{k-1}$ edges [167]. The nodes can be represented by k-bit words. A node with bit representation α is connected with a node with bit representation β, if

- α and β differ in the last bit (*exchange edge*) or
- α results from β by a cyclic left shift or a cyclic right shift (*shuffle edge*).

Figure 2.12(i) shows a shuffle–exchange network with 8 nodes. The permutation (α, β) where β results from α by a cyclic left shift is called **perfect shuffle**. The permutation (α, β) where β results from α by a cyclic right shift is called **inverse perfect shuffle**, see [115] for a detailed treatment of shuffle–exchange networks.

A k-**ary** d-**cube** with $k \geq 2$ is a generalization of the d-dimensional cube with $n = k^d$ nodes where each dimension i with $i = 0, \ldots, d-1$ contains k nodes. Each node can be represented by a word with d numbers $(a_0, \ldots, a_{d-1})$ with $0 \leq a_i \leq k-1$, where a_i represents the position of the node in dimension i, $i = 0, \ldots, d-1$. Two nodes $A = (a_0, \ldots, a_{d-1})$ and $B = (b_0, \ldots, b_{d-1})$ are connected by an edge if there is a dimension $j \in \{0, \ldots, d-1\}$ for which $a_j = (b_j \pm 1) \bmod k$ and $a_i = b_i$ for all other dimensions $i = 0, \ldots, d-1, i \neq j$. For $k = 2$, each node has one neighbor in each dimension, resulting in degree $g(G) = d$. For $k > 2$, each node has two neighbors in each dimension, resulting in degree $g(G) = 2d$. The k-ary d-cube captures some of the previously considered topologies as special case: A k-ary 1-cube is a ring with k nodes, a k-ary 2-cube is a torus with k^2 nodes, a 3-ary 3-cube is a three-dimensional torus with $3 \times 3 \times 3$ nodes, and a 2-ary d-cube is a d-dimensional cube.

Table 2.2 summarizes important characteristics of the network topologies described.

2.5.3 Embeddings

In this section, we consider the embedding of several networks into a hypercube network, demonstrating that the hypercube topology is versatile and flexible.

2.5.3.1 Embedding a Ring into a Hypercube Network

For an embedding of a ring network with $n = 2^k$ nodes represented by $V' = \{1, \ldots, n\}$ in a k-dimensional cube with nodes $V = \{0, 1\}^k$, a bijective function from V' to V is constructed such that a ring edge $(i, j) \in E'$ is mapped to a hypercube edge. In the ring, there are edges between neighboring nodes in the sequence

$1, \ldots, n$. To construct the embedding, we have to arrange the hypercube nodes in V in a sequence such that there is also an edge between neighboring nodes in the sequence. The sequence is constructed as reflected Gray code (RGC) sequence which is defined as follows:

A k-bit RGC is a sequence with 2^k binary strings of length k such that two neighboring strings differ in exactly one bit position. The RGC sequence is constructed recursively, as follows:

- The 1-bit RGC sequence is $\text{RGC}_1 = (0, 1)$.
- The 2-bit RGC sequence is obtained from RGC_1 by inserting a 0 and a 1 in front of RGC_1, resulting in the two sequences (00, 01) and (10, 11). Reversing the second sequence and concatenation yields $\text{RGC}_2 = (00, 01, 11, 10)$.
- For $k \geq 2$, the k-bit Gray code RGC_k is constructed from the $(k-1)$-bit Gray code $\text{RGC}_{k-1} = (b_1, \ldots, b_m)$ with $m = 2^{k-1}$ where each entry b_i for $1 \leq i \leq m$ is a binary string of length $k-1$. To construct RGC_k, RGC_{k-1} is duplicated; a 0 is inserted in front of each b_i of the original sequence, and a 1 is inserted in front of each b_i of the duplicated sequence. This results in sequences $(0b_1, \ldots, 0b_m)$ and $(1b_1, \ldots, 1b_m)$. RGC_k results by reversing the second sequence and concatenating the two sequences; thus $RGC_k = (0b_1, \ldots, 0b_m, 1b_m, \ldots, 1b_1)$.

The Gray code sequences RGC_k constructed in this way have the property that they contain all binary representations of a k-dimensional hypercube, since the construction corresponds to the construction of a k-dimensional cube from two $(k-1)$-dimensional cubes as described in the previous section. Two neighboring k-bit words of RGC_k differ in exactly one bit position, as can be shown by induction. The statement is surely true for RGC_1. Assuming that the statement is true for RGC_{k-1}, it is true for the first 2^{k-1} elements of RGC_k as well as for the last 2^{k-1} elements, since these differ only by a leading 0 or 1 from RGC_{k-1}. The statement is also true for the two middle elements $0b_m$ and $1b_m$ at which the two sequences of length 2^{k-1} are concatenated. Similarly, the first element $0b_1$ and the last element $1b_1$ of RGC_k differ only in the first bit. Thus, neighboring elements of RGC_k are connected by a hypercube edge.

An embedding of a ring into a k-dimensional cube can be defined by the mapping

$$\sigma : \{1, \ldots, n\} \rightarrow \{0, 1\}^k \text{ with } \sigma(i) := RGC_k(i),$$

where $\text{RGC}_k(i)$ denotes the ith element of RGC_k. Figure 2.14(a) shows an example for $k = 3$.

2.5.3.2 Embedding a Two-Dimensional Mesh into a Hypercube Network

The embedding of a two-dimensional mesh with $n = n_1 \cdot n_2$ nodes into a k-dimensional cube with $n = 2^k$ nodes can be obtained by a generalization of the embedding of a ring network. For k_1 and k_2 with $n_1 = 2^{k_1}$ and $n_2 = 2^{k_2}$, i.e., $k_1 + k_2 = k$, the Gray codes $\text{RGC}_{k_1} = (a_1, \ldots, a_{n_1})$ and $\text{RGC}_{k_2} = (b_1, \ldots, b_{n_2})$ are

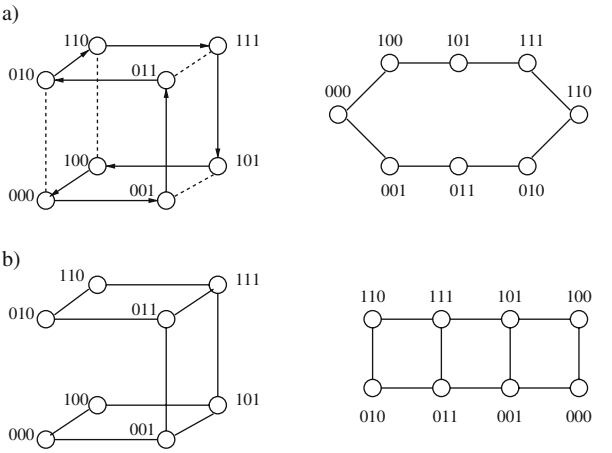


Fig. 2.14 Embeddings into a hypercube network: (**a**) embedding of a ring network with 8 nodes into a three-dimensional hypercube and (**b**) embedding of a two-dimensional 2×4 mesh into a three-dimensional hypercube

used to construct an $n_1 \times n_2$ matrix M whose entries are k-bit strings. In particular, it is

$$M = \begin{bmatrix} a_1 b_1 & a_1 b_2 & \dots & a_1 b_{n_2} \\ a_2 b_1 & a_2 b_2 & \dots & a_2 b_{n_2} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n_1} b_1 & a_{n_1} b_2 & \dots & a_{n_1} b_{n_2} \end{bmatrix}.$$

The matrix is constructed such that neighboring entries differ in exactly one bit position. This is true for neighboring elements in a row, since identical elements of RGC_{k_1} and neighboring elements of RGC_{k_2} are used. Similarly, this is true for neighboring elements in a column, since identical elements of RGC_{k_2} and neighboring elements of RGC_{k_1} are used. All elements of M are bit strings of length k and there are no identical bit strings according to the construction. Thus, the matrix M contains all bit representations of nodes in a k-dimensional cube and neighboring entries in M correspond to neighboring nodes in the k-dimensional cube, which are connected by an edge. Thus, the mapping

$$\sigma : \{1, \dots, n_1\} \times \{1, \dots, n_2\} \to \{0, 1\}^k \text{ with } \sigma(i, j) = M(i, j)$$

is an embedding of the two-dimensional mesh into the k-dimensional cube. Figure 2.14(b) shows an example.

2.5.3.3 Embedding of a d-Dimensional Mesh into a Hypercube Network

In a d-dimensional mesh with $n_i = 2^{k_i}$ nodes in dimension i, $1 \leq i \leq d$, there are $n = n_1 \cdot \dots \cdot n_d$ nodes in total. Each node can be represented by its mesh coordinates $(x_1, \dots, x_d)$ with $1 \leq x_i \leq n_i$. The mapping

$$\sigma : \{(x_1, \ldots, x_d) \mid 1 \leq x_i \leq n_i, 1 \leq i \leq d\} \longrightarrow \{0,1\}^k$$
$$\text{with } \sigma((x_1, \ldots, x_d)) = s_1 s_2 \ldots s_d \text{ and } s_i = \text{RGC}_{k_i}(x_i)$$

(where s_i is the x_ith bit string in the Gray code sequence RGC_{k_i}) defines an embedding into the k-dimensional cube. For two mesh nodes $(x_1, \ldots, x_d)$ and $(y_1, \ldots, y_d)$ that are connected by an edge in the d-dimensional mesh, there exists exactly one dimension $i \in \{1, \ldots, d\}$ with $|x_i - y_i| = 1$ and for all other dimensions $j \neq i$, it is $x_j = y_j$. Thus, for the corresponding hypercube nodes $\sigma((x_1, \ldots, x_d)) = s_1 s_2 \ldots s_d$ and $\sigma((y_1, \ldots, y_d)) = t_1 t_2 \ldots t_d$, all components $s_j = \text{RGC}_{k_j}(x_j) = \text{RGC}_{k_j}(y_j) = t_j$ for $j \neq i$ are identical. Moreover, $\text{RGC}_{k_i}(x_i)$ and $\text{RGC}_{k_i}(y_i)$ differ in exactly one bit position. Thus, the hypercube nodes $s_1 s_2 \ldots s_d$ and $t_1 t_2 \ldots t_d$ also differ in exactly one bit position and are therefore connected by an edge in the hypercube network.

2.5.4 Dynamic Interconnection Networks

Dynamic interconnection networks are also called indirect interconnection networks. In these networks, nodes or processors are not connected directly with each other. Instead, switches are used and provide an *indirect* connection between the nodes, giving these networks their name. From the processors' point of view, such a network forms an interconnection unit into which data can be sent and from which data can be received. Internally, a dynamic network consists of switches that are connected by physical links. For a message transmission from one node to another node, the switches can be configured *dynamically* such that a connection is established.

Dynamic interconnection networks can be characterized according to their topological structure. Popular forms are bus networks, multistage networks, and crossbar networks.

2.5.4.1 Bus Networks

A bus essentially consists of a set of wires which can be used to transport data from a sender to a receiver, see Fig. 2.15 for an illustration. In some cases, several hundreds

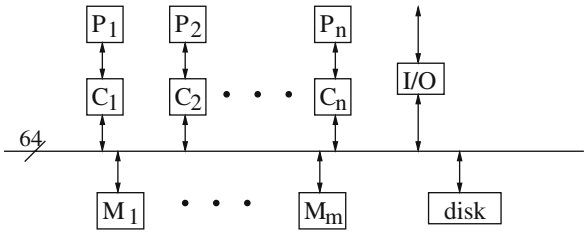


Fig. 2.15 Illustration of a bus network with 64 wires to connect processors $P_1, \ldots, P_n$ with caches $C_1, \ldots, C_n$ to memory modules $M_1, \ldots, M_m$

of wires are used to ensure a fast transport of large data sets. At each point in time, only one data transport can be performed via the bus, i.e., the bus must be used in a time-sharing way. When several processors attempt to use the bus simultaneously, a **bus arbiter** is used for the coordination. Because the likelihood for simultaneous requests of processors increases with the number of processors, bus networks are typically used for a small number of processors only.

2.5.4.2 Crossbar Networks

An $n \times m$ crossbar network has n inputs and m outputs. The actual network consists of $n \cdot m$ switches as illustrated in Fig. 2.16 (left). For a system with a shared address space, the input nodes may be processors and the outputs may be memory modules. For a system with a distributed address space, both the input nodes and the output nodes may be processors. For each request from a specific input to a specific output, a connection in the switching network is established. Depending on the specific input and output nodes, the switches on the connection path can have different states (straight or direction change) as illustrated in Fig. 2.16 (right). Typically, crossbar networks are used only for a small number of processors because of the large hardware overhead required.

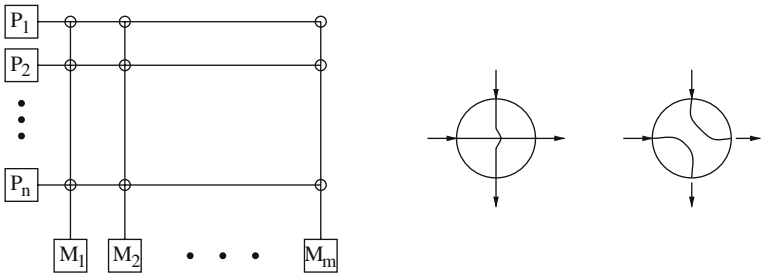


Fig. 2.16 Illustration of a $n \times m$ crossbar network for n processors and m memory modules (*left*). Each network switch can be in one of two states: straight or direction change (*right*)

2.5.4.3 Multistage Switching Networks

Multistage switching networks consist of several stages of switches with connecting wires between neighboring stages. The network is used to connect input devices to output devices. Input devices are typically the processors of a parallel system. Output devices can be processors (for distributed memory machines) or memory modules (for shared memory machines). The goal is to obtain a small distance for arbitrary pairs of input and output devices to ensure fast communication. The internal connections between the stages can be represented as a graph where switches are represented by nodes and wires between switches are represented by edges. Input and output devices can be represented as specialized nodes with edges going into

the actual switching network graph. The construction of the switching graph and the degree of the switches used are important characteristics of multistage switching networks.

Regular multistage interconnection networks are characterized by a *regular* construction method using the same degree of incoming and outgoing wires for all switches. For the switches, $a \times b$ crossbars are often used where a is the input degree and b is the output degree. The switches are arranged in stages such that neighboring stages are connected by fixed interconnections, see Fig. 2.17 for an illustration. The input wires of the switches of the first stage are connected with the input devices. The output wires of the switches of the last stage are connected with the output devices. Connections from input devices to output devices are performed by selecting a path from a specific input device to the selected output device and setting the switches on the path such that the connection is established.

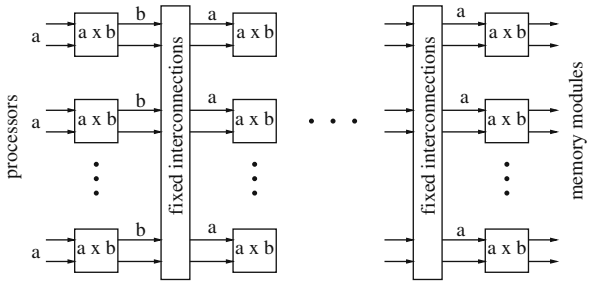


Fig. 2.17 Multistage interconnection networks with $a \times b$ crossbars as switches according to [95]

The actual graph representing a regular multistage interconnection network results from *gluing* neighboring stages of switches together. The connection between neighboring stages can be described by a directed acyclic graph of depth 1. Using w nodes for each stage, the degree of each node is $g = n/w$ where n is the number of edges between neighboring stages. The connection between neighboring stages can be represented by a permutation $\pi : \{1, \ldots, n\} \to \{1, \ldots, n\}$ which specifies which output link of one stage is connected to which input link of the next stage. This means that the output links $\{1, \ldots, n\}$ of one stage are connected to the input links $(\pi(1), \ldots, \pi(n))$ of the next stage. Partitioning the permutation $(\pi(1), \ldots, \pi(n))$ into w parts results in the ordered set of input links of nodes of the next stage. For regular multistage interconnection networks, the same permutation is used for all stages, and the stage number can be used as parameter.

Popular regular multistage networks are the omega network, the baseline network, and the butterfly network. These networks use 2×2 crossbar switches which are arranged in $\log n$ stages. Each switch can be in one of four states as illustrated in Fig. 2.18. In the following, we give a short overview of the omega, baseline, butterfly, Beneš, and fat tree networks, see [115] for a detailed description.

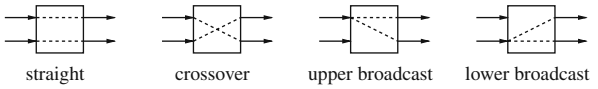


Fig. 2.18 Settings for switches in an omega, baseline, or butterfly network

2.5.4.4 Omega Network

An $n \times n$ omega network is based on 2×2 crossbar switches which are arranged in $\log n$ stages such that each stage contains $n/2$ switches where each switch has two input links and two output links. Thus, there are $(n/2) \cdot \log n$ switches in total, with $\log n \equiv \log_2 n$. Each switch can be in one of four states, see Fig. 2.18. In the omega network, the permutation function describing the connection between neighboring stages is the same for all stages, independent of the number of the stage. The switches in the network are represented by pairs (α, i) where $\alpha \in \{0, 1\}^{\log n - 1}$ is a bit string of length $\log n - 1$ representing the position of a switch within a stage and $i \in \{0, \ldots, \log n - 1\}$ is the stage number. There is an edge from node (α, i) in stage i to two nodes $(\beta, i + 1)$ in stage $i + 1$ where β is defined as follows:

1. β results from α by a cyclic left shift or
2. β results from α by a cyclic left shift followed by an inversion of the last (rightmost) bit.

An $n \times n$ omega network is also called $(\log n - 1)$-dimensional omega network. Figure 2.19(a) shows a 16×16 (three-dimensional) omega network with four stages and eight switches per stage.

2.5.4.5 Butterfly Network

Similar to the omega network, a k-dimensional butterfly network connects $n = 2^{k+1}$ inputs to $n = 2^{k+1}$ outputs using a network of 2×2 crossbar switches. Again, the switches are arranged in $k + 1$ stages with 2^k nodes/switches per stage. This results in a total number $(k + 1) \cdot 2^k$ of nodes. Again, the nodes are represented by pairs (α, i) where i for $0 \leq i \leq k$ denotes the stage number and $\alpha \in \{0, 1\}^k$ is the position of the node in the stage. The connection between neighboring stages i and $i + 1$ for $0 \leq i < k$ is defined as follows: Two nodes (α, i) and $(\alpha', i + 1)$ are connected if and only if

1. α and α' are identical (straight edge) or
2. α and α' differ in precisely the $(i + 1)$th bit from the left (cross edge).

Figure 2.19(b) shows a 16×16 butterfly network with four stages.

2.5.4.6 Baseline Network

The k-dimensional baseline network has the same number of nodes, edges, and stages as the butterfly network. Neighboring stages are connected as follows: Node (α, i) is connected to node $(\alpha', i + 1)$ for $0 \leq i < k$ if and only if

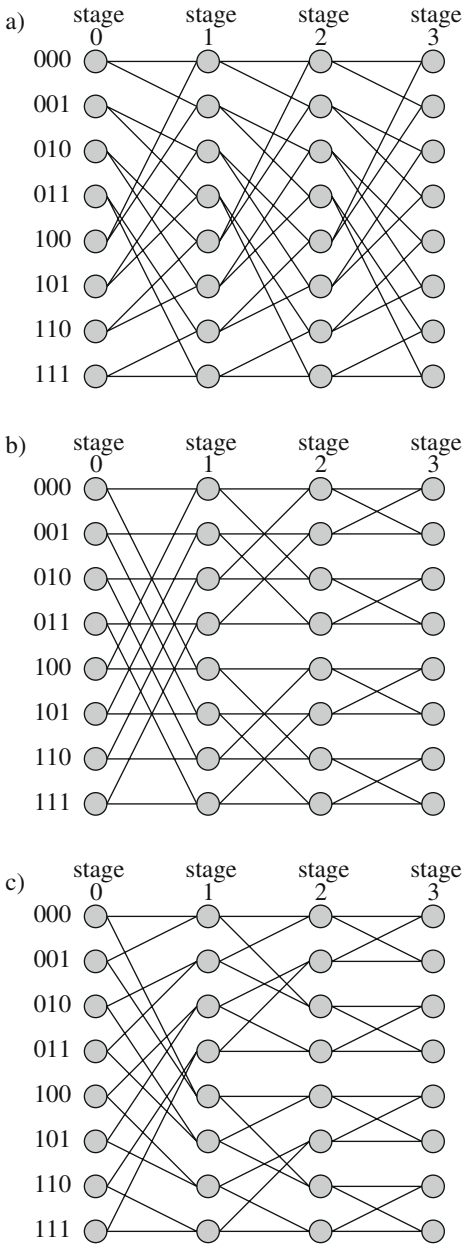


Fig. 2.19 Examples for dynamic interconnection networks: (**a**) 16×16 omega network, (**b**) 16×16 butterfly network, (**c**) 16×16 baseline network. All networks are three-dimensional

1. α' results from α by a cyclic right shift on the last $k - i$ bits of α or
2. α' results from α by first inverting the last (rightmost) bit of α and then performing a cyclic right shift on the last $k - i$ bits.

Figure 2.19(c) shows a 16×16 baseline network with four stages.

2.5.4.7 Beneš Network

The k-dimensional Beneš network is constructed from two k-dimensional butterfly networks such that the first $k + 1$ stages are a butterfly network and the last $k + 1$ stages are a reverted butterfly network. The last stage $(k + 1)$ of the first butterfly network and the first stage of the second (reverted) butterfly network are merged. In total, the k-dimensional Beneš network has $2k + 1$ stages with 2^k switches in each stage. Figure 2.20(a) shows a three-dimensional Beneš network as an example.

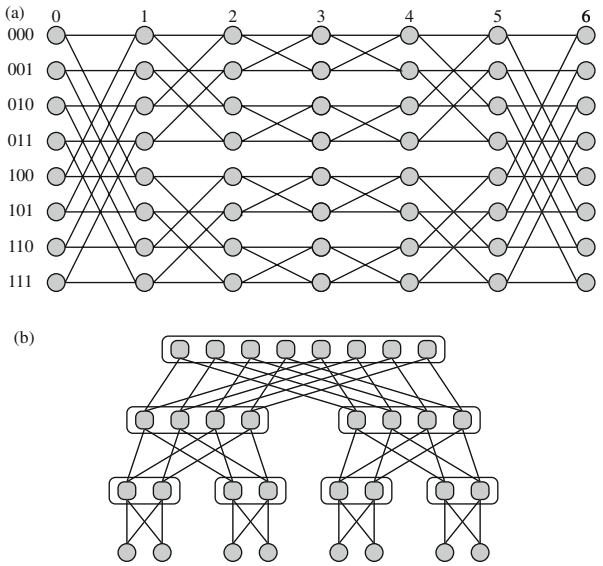


Fig. 2.20 Examples for dynamic interconnection networks: (**a**) three-dimensional Beneš network and (**b**) fat tree network for 16 processors

2.5.4.8 Fat Tree Network

The basic structure of a *dynamic tree* or *fat tree* network is a complete binary tree. The difference from a normal tree is that the number of connections between the nodes increases toward the root to avoid bottlenecks. Inner tree nodes consist of switches whose structure depends on their position in the tree structure. The leaf level is level 0. For n processors, represented by the leaves of the tree, a switch on

tree level i has 2^i input links and 2^i output links for $i = 1, \ldots, \log n$. This can be realized by assembling the switches on level i internally from 2^{i-1} switches with two input and two output links each. Thus, each level i consists of $n/2$ switches in total, grouped in $2^{\log n - i}$ nodes. This is shown in Fig. 2.20(b) for a fat tree with four layers. Only the inner switching nodes are shown, not the leaf nodes representing the processors.

2.6 Routing and Switching

Direct and indirect interconnection networks provide the physical basis to send messages between processors. If two processors are not directly connected by a network link, a path in the network consisting of a sequence of nodes has to be used for message transmission. In the following, we give a short description of how to select a suitable path in the network (routing) and how messages are handled at intermediate nodes on the path (switching).

2.6.1 Routing Algorithms

A **routing algorithm** determines a path in a given network from a source node A to a destination node B. The path consists of a sequence of nodes such that neighboring nodes in the sequence are connected by a physical network link. The path starts with node A and ends at node B. A large variety of routing algorithms have been proposed in the literature, and we can only give a short overview in the following. For a more detailed description and discussion, we refer to [35, 44].

Typically, multiple message transmissions are being executed concurrently according to the requirements of one or several parallel programs. A routing algorithm tries to reach an even load on the physical network links as well as to avoid the occurrence of deadlocks. A set of messages is in a **deadlock situation** if each of the messages is supposed to be transmitted over a link that is currently used by another message of the set. A routing algorithm tries to select a path in the network connecting nodes A and B such that minimum costs result, thus leading to a fast message transmission between A and B. The resulting communication costs depend not only on the length of the path used, but also on the load of the links on the path. The following issues are important for the path selection:

- **Network topology**: The topology of the network determines which paths are available in the network to establish a connection between nodes A and B.
- **Network contention**: Contention occurs when two or more messages should be transmitted at the same time over the same network link, thus leading to a delay in message transmission.
- **Network congestion**: Congestion occurs when too many messages are assigned to a restricted resource (like a network link or buffer) such that arriving messages

have to be discarded since they cannot be stored anywhere. Thus, in contrast to contention, congestion leads to an overflow situation with message loss [139].

A large variety of routing algorithms have been proposed in the literature. Several classification schemes can be used for a characterization. Using the path length, **minimal** and **non-minimal** routing algorithms can be distinguished. Minimal routing algorithms always select the shortest message transmission, which means that when using a link of the path selected, a message always gets closer to the target node. But this may lead to congestion situations. Non-minimal routing algorithms do not always use paths with minimum length if this is necessary to avoid congestion at intermediate nodes.

A further classification can be made by distinguishing **deterministic** routing algorithms and **adaptive** routing algorithms. A routing algorithm is deterministic if the path selected for message transmission only depends on the source and destination nodes regardless of other transmissions in the network. Therefore, deterministic routing can lead to unbalanced network load. Path selection can be done *source oriented* at the sending node or *distributed* during message transmission at intermediate nodes. An example for deterministic routing is **dimension-order routing** which can be applied for network topologies that can be partitioned into several orthogonal dimensions as is the case for meshes, tori, and hypercube topologies. Using dimension-order routing, the routing path is determined based on the position of the source node and the target node by considering the dimensions in a fixed order and traversing a link in the dimension if necessary. This can lead to network contention because of the deterministic path selection.

Adaptive routing tries to avoid such contentions by dynamically selecting the routing path based on load information. Between any pair of nodes, multiple paths are available. The path to be used is dynamically selected such that network traffic is spread evenly over the available links, thus leading to an improvement of network utilization. Moreover, *fault tolerance* is provided, since an alternative path can be used in case of a link failure. Adaptive routing algorithms can be further categorized into minimal and non-minimal adaptive algorithms as described above. In the following, we give a short overview of important routing algorithms. For a more detailed treatment, we refer to [35, 95, 44, 115, 125].

2.6.1.1 Dimension-Order Routing

We give a short description of XY routing for two-dimensional meshes and E-cube routing for hypercubes as typical examples for dimension-order routing algorithms.

XY Routing for Two-Dimensional Meshes

For a two-dimensional mesh, the position of the nodes can be described by an X-coordinate and a Y-coordinate where X corresponds to the horizontal and Y corresponds to the vertical direction. To send a message from a source node A with position (X_A, Y_A) to target node B with position (X_B, Y_B), the message is sent from

the source node into (positive or negative) X-direction until the X-coordinate X_B of B is reached. Then, the message is sent into Y-direction until Y_B is reached. The length of the resulting path is $\mid X_A - X_B \mid + \mid Y_A - Y_B \mid$. This routing algorithm is deterministic and minimal.

E-Cube Routing for Hypercubes

In a k-dimensional hypercube, each of the $n = 2^k$ nodes has a direct interconnection link to each of its k neighbors. As introduced in Sect. 2.5.2, each of the nodes can be represented by a bit string of length k such that the bit string of one of the k neighbors is obtained by inverting one of the bits in the bit string. E-cube uses the bit representation of a sending node A and a receiving node B to select a routing path between them. Let $\alpha = \alpha_0 \ldots \alpha_{k-1}$ be the bit representation of A and $\beta = \beta_0 \ldots \beta_{k-1}$ be the bit representation of B. Starting with A, in each step a dimension is selected which determines the next node on the routing path. Let A_i with bit representation $\gamma = \gamma_0 \ldots \gamma_{k-1}$ be a node on the routing path $A = A_0, A_1, \ldots, A_l = B$ from which the message should be forwarded in the next step. For the forwarding from A_i to A_{i+1}, the following two substeps are made:

- The bit string $\gamma \oplus \beta$ is computed where $\oplus$ denotes the bitwise exclusive or computation (i.e., $0 \oplus 0 = 0, 0 \oplus 1 = 1, 1 \oplus 0 = 1, 1 \oplus 1 = 0$).
- The message is forwarded in dimension d where d is the rightmost bit position of $\gamma \oplus \beta$ with value 1. The next node A_{i+1} on the routing path is obtained by inverting the dth bit in γ, i.e., the bit representation of A_{i+1} is $\delta = \delta_0 \ldots \delta_{k-1}$ with $\delta_j = \gamma_j$ for $j \neq d$ and $\delta_d = \bar{\gamma_d}$. The target node B is reached when $\gamma \oplus \beta = 0$.

Example For $k = 3$, let A with bit representation $\alpha = 010$ be the source node and B with bit representation $\beta = 111$ be the target node. First, the message is sent from A into direction $d = 2$ to A_1 with bit representation 011 (since $\alpha \oplus \beta = 101$). Then, the message is sent in dimension $d = 0$ to β since ($011 \oplus 111 = 100$).

2.6.1.2 Deadlocks and Routing Algorithms

Usually, multiple messages are in transmission concurrently. A deadlock occurs if the transmission of a subset of the messages is blocked forever. This can happen in particular if network resources can be used only by one message at a time. If, for example, the links between two nodes can be used by only one message at a time and if a link can only be released when the following link on the path is free, then the mutual request for links can lead to a deadlock. Such deadlock situations can be avoided by using a suitable routing algorithm. Other deadlock situations that occur because of limited size of the input or output buffer of the interconnection links or because of an unsuited order of the send and receive operations are considered in Sect. 2.6.3 on switching strategies and Chap. 5 on message-passing programming.

To prove the deadlock freedom of routing algorithms, possible dependencies between interconnection channels are considered. A dependence from an intercon-

nection channel l_1 to an interconnection channel l_2 exists, if it is possible that the routing algorithm selects a path which contains channel l_2 directly after channel l_1. These dependencies between interconnection channels can be represented by a **channel dependence graph** which contains the interconnection channels as nodes; each dependence between two channels is represented by an edge. A routing algorithm is deadlock free for a given topology, if the channel dependence graph does not contain cycles. In this case, no communication pattern can ever lead to a deadlock.

For topologies that do not contain cycles, no channel dependence graph can contain cycles, and therefore each routing algorithm for such a topology must be deadlock free. For topologies with cycles, the channel dependence graph must be analyzed. In the following, we show that XY routing for two-dimensional meshes with bidirectional links is deadlock free.

Deadlock Freedom of XY Routing

The channel dependence graph for XY routing contains a node for each unidirectional link of the two-dimensional $n_X \times n_Y$ mesh, i.e., there are two nodes for each bidirectional link of the mesh. There is a dependence from link u to link v, if v can be directly reached from u in horizontal or vertical direction or by a 90° (deg) turn down or up. To show the deadlock freedom, all unidirectional links of the mesh are numbered as follows:

- Each horizontal edge from node (i, y) to node $(i + 1, y)$ gets number $i + 1$ for $i = 0, \ldots, n_x - 2$ for each valid value of y. The opposite edge from $(i + 1, y)$ to (i, y) gets number $n_x - 1 - (i + 1) = n_x - i - 2$ for $i = 0, \ldots, n_x - 2$. Thus, the edges in increasing x-direction are numbered from 1 to $n_x - 1$, the edges in decreasing x-direction are numbered from 0 to $n_x - 2$.
- Each vertical edge from (x, j) to $(x, j+1)$ gets number $j+n_x$ for $j = 0, \ldots, n_y - 2$. The opposite edge from $(x, j + 1)$ to (x, j) gets number $n_x + n_y - (j + 1)$.

Figure 2.21 shows a 3×3 mesh and the resulting channel dependence graph for XY routing. The nodes of the graph are annotated with the numbers assigned to the corresponding network links. It can be seen that all edges in the channel dependence graph go from a link with a smaller number to a link with a larger number. Thus, a delay during message transmission along a routing path can occur only if the message has to wait after the transmission along a link with number i for the release of a successive link w with number $j > i$ currently used by another message transmission (delay condition). A deadlock can only occur if a set of messages $\{N_1, \ldots, N_k\}$ and network links $\{n_1, \ldots, n_k\}$ exists such that for $1 \leq i < k$ each message N_i uses a link n_i for transmission and waits for the release of link n_{i+1} which is currently used for the transmission of message N_{i+1}. Additionally, N_k is currently transmitted using link n_k and waits for the release of n_1 used by N_1. If $n()$ denotes the numbering of the network links introduced above, the delay condition implies that for the deadlock situation just described, it must be

$$n(n_1) < n(n_2) < \cdots < n(n_k) < n(n_1).$$

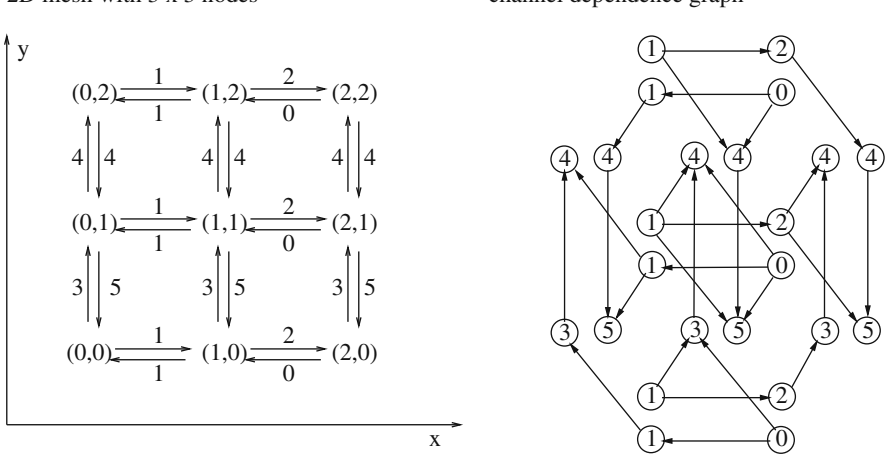


Fig. 2.21 3×3 mesh and corresponding channel dependence graph for XY routing

This is a contradiction, and thus no deadlock can occur. Each routing path selected by XY routing consists of a sequence of links with increasing numbers. Each edge in the channel dependence graph points to a link with a larger number than the source link. Thus, there can be no cycles in the channel dependence graph. A similar approach can be used to show deadlock freedom for E-cube routing, see [38].

2.6.1.3 Source-Based Routing

Source-based routing is a deterministic routing algorithm for which the source node determines the entire path for message transmission. For each node n_i on the path, the output link number a_i is determined, and the sequence of output link numbers $a_0, \ldots, a_{n-1}$ to be used is added as header to the message. When the message passes a node, the first link number is stripped from the front of the header and the message is forwarded through the specified link to the next node.

2.6.1.4 Table-Driven Routing

For table-driven routing, each node contains a routing table which contains for each destination node the output link to be used for the transmission. When a message arrives at a node, a lookup in the routing table is used to determine how the message is forwarded to the next node.

2.6.1.5 Turn Model Routing

The turn model [68, 125] tries to avoid deadlocks by a suitable selection of turns that are allowed for the routing. Deadlocks occur if the paths for message transmission contain turns that may lead to cyclic waiting in some situations. Deadlocks can

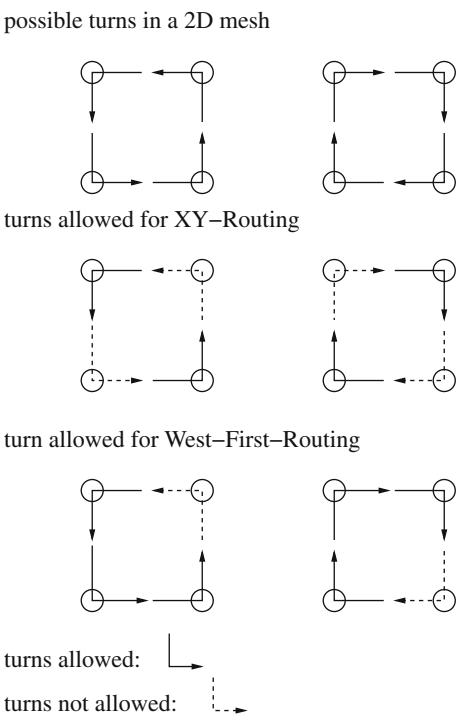


Fig. 2.22 Illustration of turns for a two-dimensional mesh with all possible turns (*top*), allowed turns for XY routing (*middle*), and allowed turns for west-first routing (*bottom*)

be avoided by prohibiting some of the turns. An example is the XY routing on a two-dimensional mesh. From the eight possible turns, see Fig. 2.22 (top), only four are allowed for XY routing, prohibiting turns from vertical into horizontal direction, see Fig. 2.22 (middle) for an illustration. The remaining four turns are not allowed in order to prevent cycles in the networks. This not only avoids the occurrence of deadlocks, but also prevents the use of adaptive routing. For n-dimensional meshes and, in the general case, k-ary d-cubes, the turn model tries to identify a minimum number of turns that must be prohibited for routing paths to avoid the occurrence of cycles. Examples are the west-first routing for two-dimensional meshes and the P-cube routing for n-dimensional hypercubes.

The **west-first routing** algorithm for a two-dimensional mesh prohibits only two of the eight possible turns: Turns to the west (left) are prohibited, and only the turns shown in Fig. 2.22 (bottom) are allowed. Routing paths are selected such that messages that must travel to the west must do so before making any turns. Such messages are sent to the west first until the requested x-coordinate is reached. Then the message can be adaptively forwarded to the south (bottom), east (right), or north (top). Figure 2.23 shows some examples for possible routing paths [125]. West-first routing is deadlock free, since cycles are avoided. For the selection of minimal routing paths, the algorithm is adaptive only if the target node lies to the east (right). Using non-minimal routing paths, the algorithm is always adaptive.

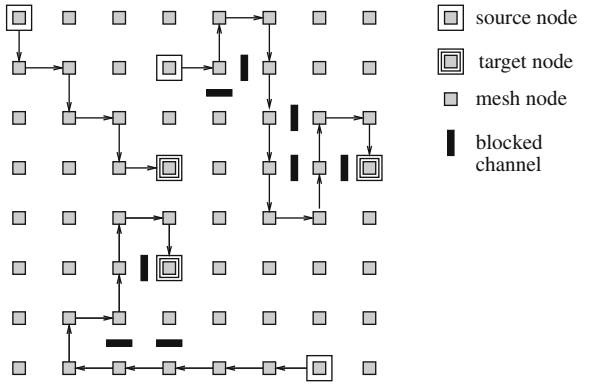


Fig. 2.23 Illustration of path selection for west-first routing in an 8×8 mesh. The links shown as blocked are used for other message transmissions and are not available for the current transmission. One of the paths shown is minimal, the other two are non-minimal, since some of the links are blocked

Routing in the n-dimensional hypercube can be done with **P-cube routing**. To send a message from a sender A with bit representation $\alpha = \alpha_0 \ldots \alpha_{n-1}$ to a receiver B with bit representation $\beta = \beta_0 \ldots \beta_{n-1}$, the bit positions in which α and β differ are considered. The number of these bit positions is the Hamming distance between A and B which determines the minimum length of a routing path from A to B. The set $E = \{i \mid \alpha_i \neq \beta_i, i = 0, \ldots, n-1\}$ of different bit positions is partitioned into two sets $E_0 = \{i \in E \mid \alpha_i = 0 \text{ and } \beta_i = 1\}$ and $E_1 = \{i \in E \mid \alpha_i = 1 \text{ and } \beta_i = 0\}$. Message transmission from A to B is split into two phases accordingly: First, the message is sent into the dimensions in E_0 and then into the dimensions in E_1.

2.6.1.6 Virtual Channels

The concept of *virtual channels* is often used for minimal adaptive routing algorithms. To provide multiple (virtual) channels between neighboring network nodes, each physical link is split into multiple virtual channels. Each virtual channel has its own separate buffer. The provision of virtual channels does not increase the number of physical links in the network, but can be used for a systematic avoidance of deadlocks.

Based on virtual channels, a network can be split into several virtual networks such that messages injected into a virtual network can only move in one direction for each dimension. This can be illustrated for a two-dimensional mesh which is split into two virtual networks, a $+X$ network and a $-X$ network, see Fig. 2.24 for an illustration. Each virtual network contains all nodes, but only a subset of the virtual channels. The $+X$ virtual network contains in the vertical direction all virtual channels between neighboring nodes, but in the horizontal direction only the virtual channels in positive direction. Similarly, the $-X$ virtual network contains in the horizontal direction only the virtual channels in negative direction, but all virtual channels in the vertical direction. The latter is possible by the definition of a suitable number of virtual channels in the vertical direction. Messages from a node A with x-coordinate x_A to a node B with x-coordinate x_B are sent in the $+X$ network, if $x_A < x_B$. Messages from A to B with $x_A > x_B$ are sent in the $-X$ network. For

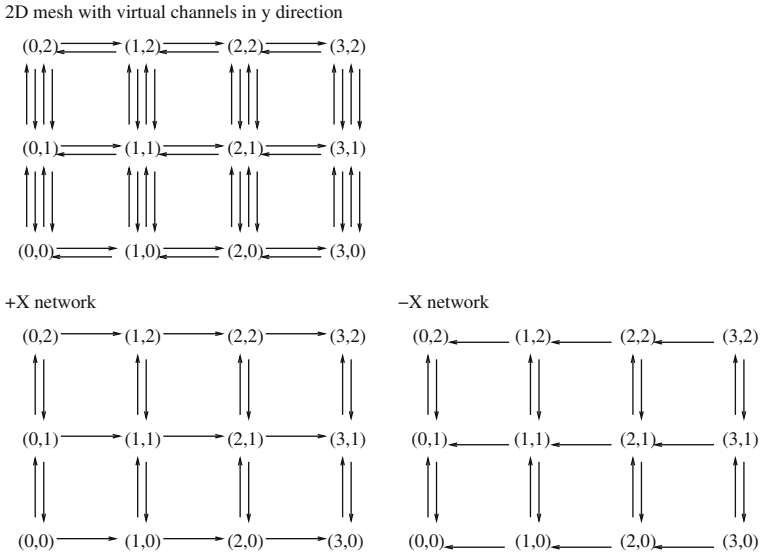


Fig. 2.24 Partitioning of a two-dimensional mesh with virtual channels into a $+X$ network and a $-X$ network for applying a minimal adaptive routing algorithm

$x_A = x_B$, one of the two networks can be selected arbitrarily, possibly using load information for the selection. The resulting adaptive routing algorithm is deadlock free [125]. For other topologies like hypercubes or tori, more virtual channels might be needed to provide deadlock freedom [125].

A non-minimal adaptive routing algorithm can send messages over longer paths if no minimal path is available. **Dimension reversal routing** can be applied to arbitrary meshes and k-ary d-cubes. The algorithm uses r pairs of virtual channels between any pair of nodes that is connected by a physical link. Correspondingly, the network is split into r virtual networks where network i for $i = 0, \ldots, r-1$ uses all virtual channels i between the nodes. Each message to be transmitted is assigned a class c with initialization $c = 0$ which can be increased to $c = 1, \ldots, r-1$ during message transmission. A message with class $c = i$ can be forwarded in network i in each dimension, but the dimensions must be traversed in increasing order. If a message must be transmitted in opposite order, its class is increased by 1 (reverse dimension order). The parameter r controls the number of dimension reversals that are allowed. If $c = r$ is reached, the message is forwarded according to dimension-ordered routing.

2.6.2 Routing in the Omega Network

The omega network introduced in Sect. 2.5.4 allows message forwarding using a distributed algorithm where each switch can forward the message without

coordination with other switches. For the description of the algorithm, it is useful to represent each of the n input channels and output channels by a bit string of length $\log n$ [115]. To forward a message from an input channel with bit representation α to an output channel with bit representation β the receiving switch on stage k of the network, $k = 0, \ldots, \log n - 1$, considers the kth bit β_k (from the left) of β and selects the output link for forwarding the message according to the following rule:

- for $\beta_k = 0$, the message is forwarded over the upper link of the switch and
- for $\beta_k = 1$, the message is forwarded over the lower link of the switch.

Figure 2.25 illustrates the path selected for message transmission from input channel $\alpha = 010$ to the output channel $\beta = 110$ according to the algorithm just described. In an $n \times n$ omega network, at most n messages from different input channels to different output channels can be sent concurrently without collision. An example of a concurrent transmission of $n = 8$ messages in an 8×8 omega network can be described by the permutation

$$\pi^8 = \begin{pmatrix} 0\,1\,2\,3\,4\,5\,6\,7 \\ 7\,3\,0\,1\,2\,5\,4\,6 \end{pmatrix},$$

which specifies that the messages are sent from input channel i ($i = 0, \ldots, 7$) to output channel $\pi^8(i)$. The corresponding paths and switch positions for the eight paths are shown in Fig. 2.26.

Many simultaneous message transmissions that can be described by permutations $\pi^8 : \{0, \ldots, n-1\} \rightarrow \{0, \ldots, n-1\}$ cannot be executed concurrently since **network conflicts** would occur. For example, the two message transmissions from $\alpha_1 = 010$ to $\beta_1 = 110$ and from $\alpha_2 = 000$ to $\beta_2 = 111$ in an 8×8 omega network would lead to a conflict. These kinds of conflicts occur, since there is exactly one path for any pair (α, β) of input and output channels, i.e., there is no alternative to avoid a critical switch. Networks with this characteristic are also called **blocking networks**. Conflicts in blocking networks can be resolved by multiple transmissions through the network.

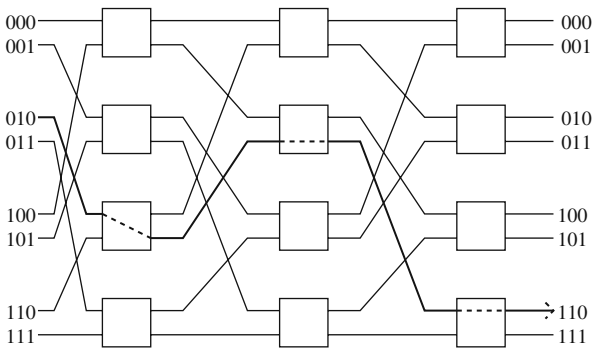


Fig. 2.25 8×8 omega network with path from 010 to 110 [14]

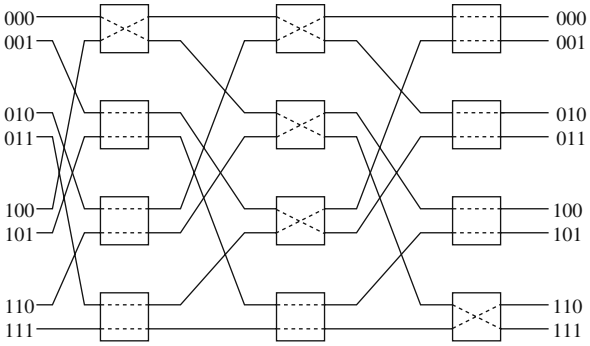


Fig. 2.26 8×8 omega network with switch positions for the realization of π^8 from the text

There is a notable number of permutations that cannot be implemented in one switching of the network. This can be seen as follows. For the connection from the n input channels to the n output channels, there are in total $n!$ possible permutations, since each output channel must be connected to exactly one input channel. There are in total $n/2 \cdot \log n$ switches in the omega network, each of which can be in one of two positions. This leads to $2^{n/2 \cdot \log n} = n^{n/2}$ different switchings of the entire network, corresponding to n concurrent paths through the network. In conclusion, only $n^{n/2}$ of the $n!$ possible permutations can be performed without conflicts.

Other examples for blocking networks are the butterfly or banyan network, the baseline network, and the delta network [115]. In contrast, the Beneš network is a non-blocking network since there are different paths from an input channel to an output channel. For each permutation $\pi : \{0, \ldots, n-1\} \to \{0, \ldots, n-1\}$ there exists a switching of the Beneš network which realizes the connection from input i to output $\pi(i)$ for $i = 0, \ldots, n-1$ concurrently without collision, see [115] for more details. As example, the switching for the permutation

$$\pi^8 = \begin{pmatrix} 0\,1\,2\,3\,4\,5\,6\,7 \\ 5\,3\,4\,7\,0\,1\,2\,6 \end{pmatrix}$$

is shown in Fig. 2.27.

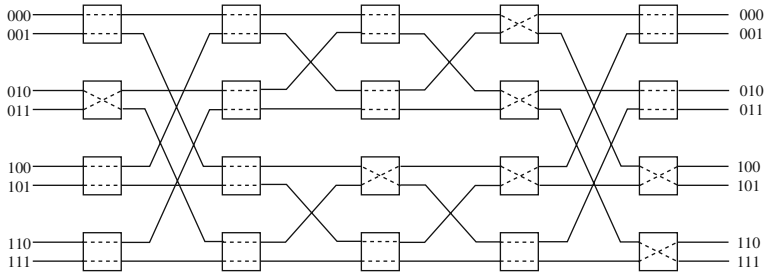


Fig. 2.27 8×8 Beneš network with switch positions for the realization of π^8 from the text

2.6.3 Switching

The switching strategy determines how a message is transmitted along a path that has been selected by the routing algorithm. In particular, the switching strategy determines

- whether and how a message is split into pieces, which are called packets or *flits* (for *flow control units*),
- how the transmission path from the source node to the destination node is allocated, and
- how messages or pieces of messages are forwarded from the input channel to the output channel of a switch or a router. The routing algorithm only determines *which* output channel should be used.

The switching strategy may have a large influence on the message transmission time from a source to a destination. Before considering specific switching strategies, we first consider the time for message transmission between two nodes that are directly connected by a physical link.

2.6.3.1 Message Transmission Between Neighboring Processors

Message transmission between two directly connected processors is implemented as a series of steps. These steps are also called *protocol*. In the following, we sketch a simple example protocol [84]. To send a message, the sending processor performs the following steps:

1. The message is copied into a system buffer.
2. A checksum is computed and a *header* is added to the message, containing the checksum as well as additional information related to the message transmission.
3. A timer is started and the message is sent out over the network interface.

To receive a message, the receiving processor performs the following steps:

1. The message is copied from the network interface into a system buffer.
2. The checksum is computed over the data contained. This checksum is compared with the checksum stored in the header. If both checksums are identical, an acknowledgment message is sent to the sender. In case of a mismatch of the checksums, the message is discarded. The message will be re-sent again after the sender timer has elapsed.
3. If the checksums are identical, the message is copied from the system buffer into the user buffer, provided by the application program. The application program gets a notification and can continue execution.

After having sent out the message, the sending processor performs the following steps:

1. If an acknowledgment message arrives for the message sent out, the system buffer containing a copy of the message can be released.

2. If the timer has elapsed, the message will be re-sent again. The timer is started again, possibly with a longer time.

In this protocol, it has been assumed that the message is kept in the system buffer of the sender to be re-sent if necessary. If message loss is tolerated, no re-sent is necessary and the system buffer of the sender can be re-used as soon as the packet has been sent out. Message transmission protocols used in practice are typically much more complicated and may take additional aspects like network contention or possible overflows of the system buffer of the receiver into consideration. A detailed overview can be found in [110, 139].

The time for a message transmission consists of the actual transmission time over the physical link and the time needed for the software overhead of the protocol, both at the sender and the receiver side. Before considering the transmission time in more detail, we first review some performance measures that are often used in this context, see [84, 35] for more details.

- The **bandwidth** of a network link is defined as the maximum frequency at which data can be sent over the link. The bandwidth is measured in bits per second or bytes per second.
- The **byte transfer time** is the time which is required to transmit a single byte over a network link. If the bandwidth is measured in bytes per second, the byte transfer time is the reciprocal of the bandwidth.
- The **time of flight**, also referred to as *channel propagation delay*, is the time which the first bit of a message needs to arrive at the receiver. This time mainly depends on the physical distance between the sender and the receiver.
- The **transmission time** is the time needed to transmit the message over a network link. The transmission time is the message size in bytes divided by the bandwidth of the network link, measured in bytes per second. The transmission time does not take conflicts with other messages into consideration.
- The **transport latency** is the total time needed to transfer a message over a network link. This is the sum of the transmission time and the time of flight, capturing the entire time interval from putting the first bit of the message onto the network link at the sender and receiving the last bit at the receiver.
- The **sender overhead**, also referred to as *startup time*, is the time that the sender needs for the preparation of message transmission. This includes the time for computing the checksum, appending the header, and executing the routing algorithm.
- The **receiver overhead** is the time that the receiver needs to process an incoming message, including checksum comparison and generation of an acknowledgment if required by the specific protocol.
- The **throughput** of a network link is the effective bandwidth experienced by an application program.

Using these performance measures, the total latency $T(m)$ of a message of size m can be expressed as

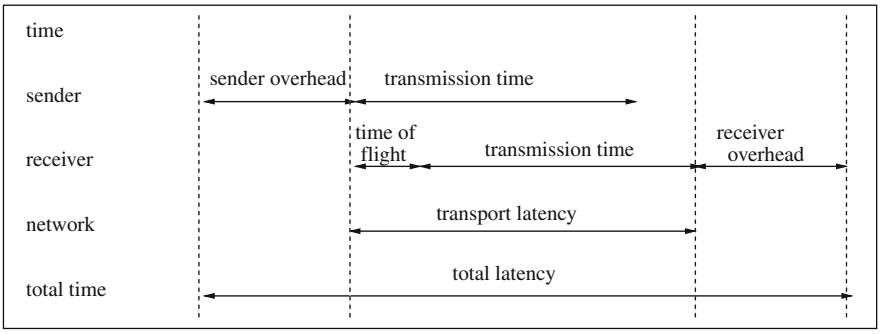


Fig. 2.28 Illustration of performance measures for the point-to-point transfer between neighboring nodes, see [84]

$$T(m) = O_{\text{send}} + T_{\text{delay}} + m/B + O_{\text{recv}}, \tag{2.1}$$

where O_{send} and O_{recv} are the sender and receiver overheads, respectively, T_{delay} is the time of flight, and B is the bandwidth of the network link. This expression does not take into consideration that a message may need to be transmitted multiple times because of checksum errors, network contention, or congestion.

The performance parameters introduced are illustrated in Fig. 2.28. Equation (2.1) can be reformulated by combining constant terms, yielding

$$T(m) = T_{\text{overhead}} + m/B \tag{2.2}$$

with $T_{\text{overhead}} = T_{\text{send}} + T_{\text{recv}}$. Thus, the latency consists of an overhead which does not depend on the message size and a term which linearly increases with the message size. Using the byte transfer time $t_B = 1/B$, Eq. (2.2) can also be expressed as

$$T(m) = T_{\text{overhead}} + t_B \cdot m. \tag{2.3}$$

This equation is often used to describe the message transmission time over a network link. When transmitting a message between two nodes that are not directly connected in the network, the message must be transmitted along a path between the two nodes. For the transmission along the path, several switching techniques can be used, including circuit switching, packet switching with store-and-forward routing, virtual cut-through routing, and wormhole routing. We give a short overview in the following.

2.6.3.2 Circuit Switching

The two basic switching strategies are circuit switching and packet switching, see [35, 84] for a detailed treatment. In **circuit switching**, the entire path from the source node to the destination node is established and reserved until the end of the transmission of this message. This means that the path is established exclusively for this

message by setting the switches or routers on the path in a suitable way. Internally, the message can be split into pieces for the transmission. These pieces can be so-called *physical units* (*phits*) denoting the amount of data that can be transmitted over a network link in one cycle. The size of the phits is determined by the number of bits that can be transmitted over a physical channel in parallel. Typical phit sizes lie between 1 bit and 256 bits. The transmission path for a message can be established by using short *probe messages* along the path. After the path is established, all phits of the message are transmitted over this path. The path can be released again by a message trailer or by an acknowledgment message from the receiver to the sender.

Sending a control message along a path of length l takes time $l \cdot t_c$ where t_c is the time to transmit the control message over a single network link. If m_c is the size of the control message, it is $t_c = t_B \cdot m_c$. After the path has been established, the transmission of the actual message of size m takes time $m \cdot t_B$. Thus, the total time of message transmission along a path of length l with circuit switching is

$$T_{cs}(m, l) = T_{\text{overhead}} + t_c \cdot l + t_B \cdot m. \tag{2.4}$$

If m_c is small compared to m, this can be reduced to $T_{\text{overhead}} + t_B \cdot m$ which is linear in m, but independent of l. Message transfer with circuit switching is illustrated in Fig. 2.30(a).

2.6.3.3 Packet Switching

For **packet switching** the message to be transmitted is partitioned into a sequence of packets which are transferred independently of each other through the network from the sender to the receiver. Using an adaptive routing algorithm, the packets can be transmitted over different paths. Each packet consists of three parts: (i) a header, containing routing and control information; (ii) the data part, containing a part of the original message; and (iii) a trailer which may contain an error control code. Each packet is sent separately to the destination according to the routing information contained in the packet. Figure 2.29 illustrates the partitioning of a message into packets. The network links and buffers are used by one packet at a time.

Packet switching can be implemented in different ways. Packet switching with **store-and-forward routing** sends a packet along a path such that the entire packet

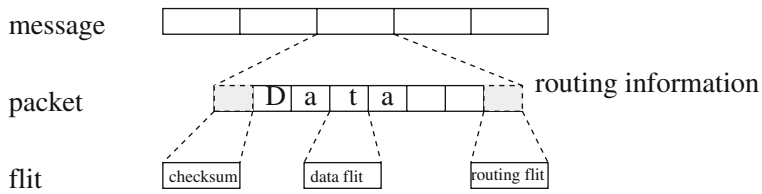


Fig. 2.29 Illustration of the partitioning of a message into packets and of packets into *flits* (*flow control units*)

is received by each switch on the path (*store*) before it is sent to the next switch on the path (*forward*). The connection between two switches A and B on the path is released for reuse by another packet as soon as the packet has been stored at B. This strategy is useful if the links connecting the switches on a path have different bandwidths as this is typically the case in *wide area networks* (WANs). In this case, store-and-forward routing allows the utilization of the full bandwidth for every link on the path. Another advantage is that a link on the path can be quickly released as soon as the packet has passed the links, thus reducing the danger of deadlocks. The drawback of this strategy is that the packet transmission time increases with the number of switches that must be traversed from source to destination. Moreover, the entire packet must be stored at each switch on the path, thus increasing the memory demands of the switches.

The time for sending a packet of size m over a single link takes time $t_h + t_B \cdot m$ where t_h is the constant time needed at each switch to store the packet in a receive buffer and to select the output channel to be used by inspecting the header information of the packet. Thus, for a path of length l, the entire time for packet transmission with store-and-forward routing is

$$T_{sf}(m, l) = t_S + l(t_h + t_B \cdot m). \tag{2.5}$$

Since t_h is typically small compared to the other terms, this can be reduced to $T_{sf}(m, l) \approx t_S + l \cdot t_B \cdot m$. Thus, the time for packet transmission depends linearly on the packet size and the length l of the path. Packet transmission with store-and-forward routing is illustrated in Fig. 2.30(b). The time for the transmission of an entire message, consisting of several packets, depends on the specific routing algorithm used. When using a deterministic routing algorithm, the message transmission time is the sum of the transmission time of all packets of the message, if no network delays occur. For adaptive routing algorithms, the transmission of the individual packets can be overlapped, thus potentially leading to a smaller message transmission time.

If all packets of a message are transmitted along the same path, **pipelining** can be used to reduce the transmission time of messages: Using pipelining, the packets of a message are sent along a path such that the links on the path are used by successive packets in an overlapping way. Using pipelining for a message of size m and packet size m_p, the time of message transmission along a path of length l can be described by

$$t_S + (m - m_p)t_B + l(t_h + t_B \cdot m_p) \approx t_S + m \cdot t_B + (l - 1)t_B \cdot m_p, \tag{2.6}$$

where $l(t_h + t_B \cdot m_p)$ is the time that elapses before the first packet arrives at the destination node. After this time, a new packet arrives at the destination in each time step of size $m_p \cdot t_B$, assuming the same bandwidth for each link on the path.

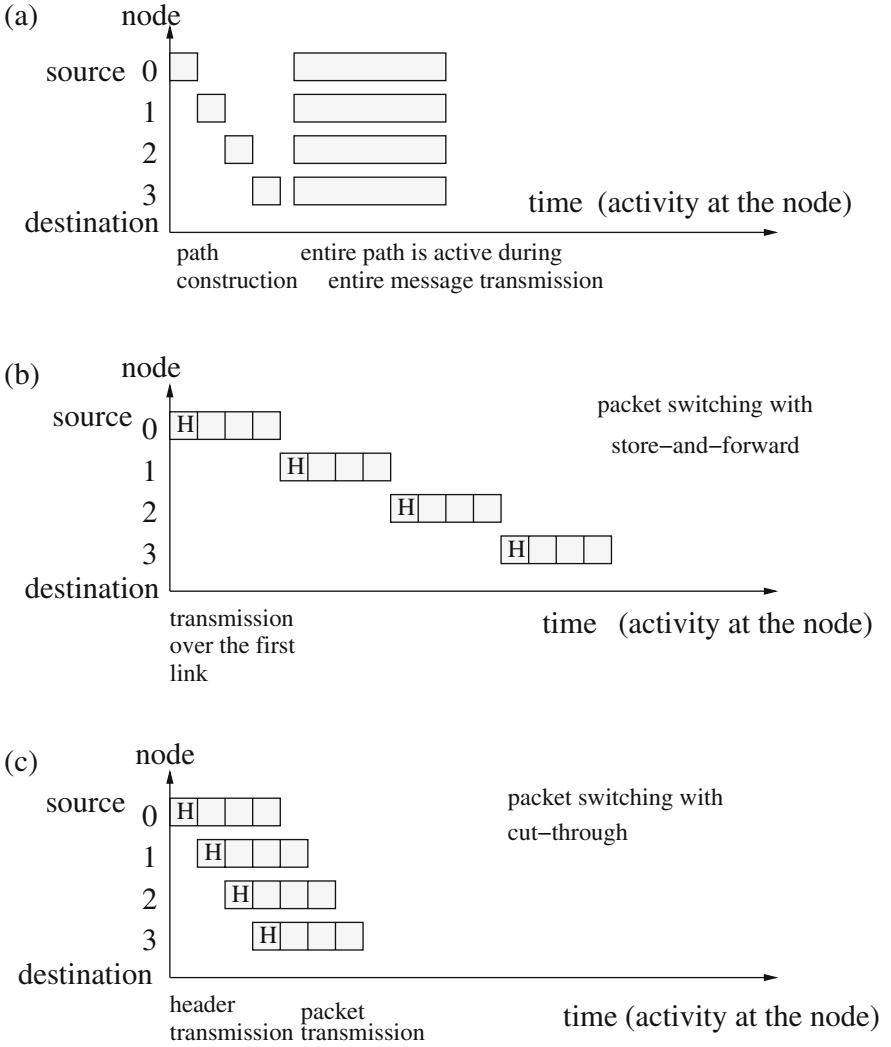


Fig. 2.30 Illustration of the latency of a point-to-point transmission along a path of length $l = 4$ for (**a**) circuit switching, (**b**) packet switching with store and forward, and (**c**) packet switching with cut-through

2.6.3.4 Cut-Through Routing

The idea of the pipelining of message packets can be extended by applying pipelining to the individual packets. This approach is taken by **cut-through routing**. Using this approach, a message is again split into packets as required by the packet-switching approach. The different packets of a message can take different paths through the network to reach the destination. Each individual packet is sent through the network in a pipelined way. To do so, each switch on the path inspects the first

few *phits* (*physical units*) of the packet header, containing the routing information, and then determines over which output channel the packet is forwarded. Thus, the transmission path of a packet is established by the packet header and the rest of the packet is transmitted along this path in a pipelined way. A link on this path can be released as soon as all *phits* of the packet, including a possible trailer, have been transmitted over this link.

The time for transmitting a header of size m_H along a single link is given by $t_H = t_B \cdot m_H$. The time for transmitting the header along a path of length l is then given by $t_H \cdot l$. After the header has arrived at the destination node, the additional time for the arrival of the rest of the packet of size m is given by $t_B(m - m_H)$. Thus, the time for transmitting a packet of size m along a path of length l using packet switching with cut-through routing can be expressed as

$$T_{ct}(m, l) = t_S + l \cdot t_H + t_B \cdot (m - m_H) \; . \tag{2.7}$$

If m_H is small compared to the packet size m, this can be reduced to $T_{ct}(m, l) \approx t_S + t_B \cdot m$. If all packets of a message use the same transmission path, and if packet transmission is also pipelined, this formula can also be used to describe the transmission time of the entire message. Message transmission time using packet switching with cut-through routing is illustrated in Fig. 2.30(c).

Until now, we have considered the transmission of a single message or packet through the network. If multiple transmissions are performed concurrently, network contention may occur because of conflicting requests to the same links. This increases the communication time observed for the transmission. The switching strategy must react appropriately if contention happens on one of the links of a transmission path. Using store-and-forward routing, the packet can simply be buffered until the output channel is free again.

With cut-through routing, two popular options are available: *virtual cut-through routing* and *wormhole routing*. Using **virtual cut-through routing**, in case of a blocked output channel at a switch, all phits of the packet in transmission are collected in a buffer at the switch until the output channel is free again. If this happens at every switch on the path, cut-through routing degrades to store-and-forward routing. Using *partial cut-through routing*, the transmission of the buffered phits of a packet can continue as soon as the output channel is free again, i.e., not all phits of a packet need to be buffered.

The **wormhole routing** approach is based on the definition of *flow control units (flits)* which are usually at least as large as the packet header. The header *flit* establishes the path through the network. The rest of the flits of the packet follow in a pipelined way on the same path. In case of a blocked output channel at a switch, only a few flits are stored at this switch, the rest is kept on the preceding switches of the path. Therefore, a blocked packet may occupy buffer space along an entire path or at least a part of the path. Thus, this approach has some similarities to circuit switching at packet level. Storing the *flits* of a blocked message along the switches of a path may cause other packets to block, leading to network saturation. Moreover, deadlocks may occur because of cyclic waiting, see Fig. 2.31 [125, 158]. An

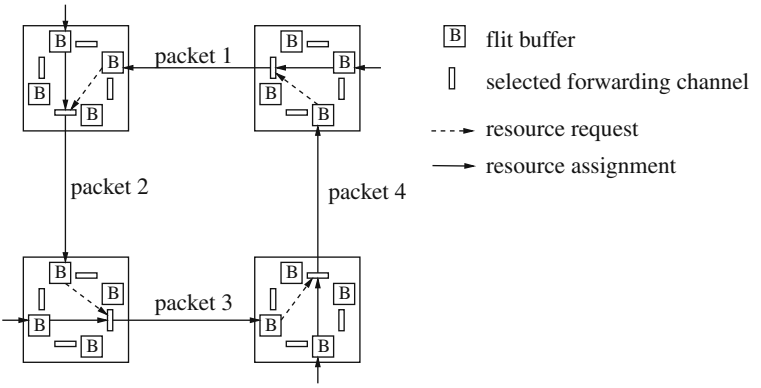


Fig. 2.31 Illustration of a deadlock situation with wormhole routing for the transmission of four packets over four switches. Each of the packets occupies a flit buffer and requests another flit buffer at the next switch; but this flit buffer is already occupied by another packet. A deadlock occurs, since none of the packets can be transmitted to the next switch

advantage of the wormhole routing approach is that the buffers at the switches can be kept small, since they need to store only a small portion of a packet.

Since buffers at the switches can be implemented large enough with today's technology, virtual cut-through routing is the more commonly used switching technique [84]. The danger of deadlocks can be avoided by using suitable routing algorithms like dimension-ordered routing or by using virtual channels, see Sect. 2.6.1.

2.6.4 Flow Control Mechanisms

A general problem in network may arise form the fact that multiple messages can be in transmission at the same time and may attempt to use the same network links at the same time. If this happens, some of the message transmissions must be blocked while others are allowed to proceed. Techniques to coordinate concurrent message transmissions in networks are called *flow control mechanisms*. Such techniques are important in all kinds of networks, including local and wide area networks, and popular protocols like TCP contain sophisticated mechanisms for flow control to obtain a high effective network bandwidth, see [110, 139] for more details. Flow control is especially important for networks of parallel computers, since these must be able to transmit a large number of messages fast and reliably. A loss of messages cannot be tolerated, since this would lead to errors in the parallel program currently executed.

Flow control mechanisms typically try to avoid congestion in the network to guarantee fast message transmission. An important aspect is the flow control mechanisms at the link level which considers message or packet transmission over a single link of the network. The link connects two switches A and B. We assume that a packet should be transmitted from A to B. If the link between A and B is

free, the packet can be transferred from the output port of A to the input port of B from which it is forwarded to the suitable output port of B. But if B is busy, there might be the situation that B does not have enough buffer space in the input port available to store the packet from A. In this case, the packet must be retained in the output buffer of A until there is enough space in the input buffer of B. But this may cause back pressure to switches preceding A, leading to the danger of network congestion. The idea of link-level flow control mechanisms is that the receiving switch provides a feedback to the sending switch, if enough input buffer space is not available, to prevent the transmission of additional packets. This feedback rapidly propagates backward in the network until the original sending node is reached. The sender can then reduce its transmission rate to avoid further packet delays.

Link-level flow control can help to reduce congestion, but the feedback propagation might be too slow and the network might already be congested when the original sender is reached. An *end-to-end flow control* with a direct feedback to the original sender may lead to a faster reaction. A windowing mechanism as used by the TCP protocol is one possibility for implementation. Using this mechanism, the sender is provided with the available buffer space at the receiver and can adapt the number of packets sent such that no buffer overflow occurs. More information can be found in [110, 139, 84, 35].

2.7 Caches and Memory Hierarchy

A significant characteristic of the hardware development during the last decades has been the increasing gap between processor cycle time and main memory access time, see Sect. 2.1. The main memory is constructed based on **DRAM** (dynamic random access memory). A typical DRAM chip has a memory access time between 20 and 70 ns whereas a 3 GHz processor, for example, has a cycle time of 0.33 ns, leading to 60–200 cycles for a main memory access. To use processor cycles efficiently, a memory hierarchy is typically used, consisting of multiple levels of memories with different sizes and access times. Only the main memory on the top of the hierarchy is built from DRAM, the other levels use **SRAM** (static random access memory), and the resulting memories are often called **caches**. SRAM is significantly faster than DRAM, but has a smaller capacity per unit area and is more costly. When using a memory hierarchy, a data item can be loaded from the fastest memory in which it is stored. The goal in the design of a memory hierarchy is to be able to access a large percentage of the data from a fast memory, and only a small fraction of the data from the slow main memory, thus leading to a small *average memory access time*.

The simplest form of a memory hierarchy is the use of a single cache between the processor and main memory (one-level cache, L1 cache). The cache contains a subset of the data stored in the main memory, and a replacement strategy is used to bring new data from the main memory into the cache, replacing data elements that are no longer accessed. The goal is to keep those data elements in the cache

which are currently used most. Today, two or three levels of cache are used for each processor, using a small and fast L1 cache and larger, but slower L2 and L3 caches.

For multiprocessor systems where each processor uses a separate local cache, there is the additional problem of keeping a consistent view of the shared address space for all processors. It must be ensured that a processor accessing a data element always accesses the most recently written data value, also in the case that another processor has written this value. This is also referred to as **cache coherence problem** and will be considered in more detail in Sect. 2.7.3.

For multiprocessors with a shared address space, the top level of the memory hierarchy is the shared address space that can be accessed by each of the processors. The design of a memory hierarchy may have a large influence on the execution time of parallel programs, and memory accesses should be ordered such that a given memory hierarchy is used as efficiently as possible. Moreover, techniques to keep a memory hierarchy consistent may also have an important influence. In this section, we therefore give an overview of memory hierarchy design and discuss issues of cache coherence and memory consistency. Since caches are the building blocks of memory hierarchies and have a significant influence on the memory consistency, we give a short overview of caches in the following subsection. For a more detailed treatment, we refer to [35, 84, 81, 137].

2.7.1 Characteristics of Caches

A cache is a small, but fast memory between the processor and the main memory. Caches are built with SRAM. Typical access times are 0.5–2.5 ns (ns = nanoseconds = 10^{-9} seconds) compared to 50–70 ns for DRAM (values from 2008 [84]). In the following, we consider a one-level cache first. A cache contains a copy of a subset of the data in main memory. Data is moved in blocks, containing a small number of words, between the cache and main memory, see Fig. 2.32. These blocks of data are called **cache blocks** or **cache lines**. The size of the cache lines is fixed for a given architecture and cannot be changed during program execution.

Cache control is decoupled from the processor and is performed by a separate cache controller. During program execution, the processor specifies memory addresses to be read or to be written as given by the load and store operations of the machine program. The processor forwards the memory addresses to the memory system and waits until the corresponding values are returned or written. The processor specifies memory addresses independently of the organization of the

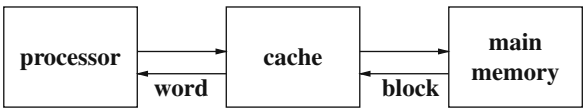


Fig. 2.32 Data transport between cache and main memory is done by the transfer of memory blocks comprising several words whereas the processor accesses single words in the cache

memory system, i.e., the processor does not need to know the architecture of the memory system. After having received a memory access request from the processor, the cache controller checks whether the memory address specified belongs to a cache line which is currently stored in the cache. If this is the case, a **cache hit** occurs, and the requested word is delivered to the processor from the cache. If the corresponding cache line is not stored in the cache, a **cache miss** occurs, and the cache line is first copied from main memory into the cache before the requested word is delivered to the processor. The corresponding delay time is also called **miss penalty**. Since the access time to main memory is significantly larger than the access time to the cache, a cache miss leads to a delay of operand delivery to the processor. Therefore, it is desirable to reduce the number of cache misses as much as possible.

The exact behavior of the cache controller is hidden from the processor. The processor observes that some memory accesses take longer than others, leading to a delay in operand delivery. During such a delay, the processor can perform other operations that are independent of the delayed operand. This is possible, since the processor is not directly occupied for the operand access from the memory system. Techniques like *operand prefetch* can be used to support an anticipated loading of operands so that other independent operations can be executed, see [84].

The number of cache misses may have a significant influence on the resulting runtime of a program. If many memory accesses lead to cache misses, the processor may often have to wait for operands, and program execution may be quite slow. Since cache management is implemented in hardware, the programmer cannot directly specify which data should reside in the cache at which point in program execution. But the order of memory accesses in a program can have a large influence on the resulting runtime, and a reordering of the memory accesses may lead to a significant reduction of program execution time. In this context, the **locality of memory accesses** is often used as a characterization of the memory accesses of a program. Spatial and temporal locality can be distinguished as follows:

- The memory accesses of a program have a high **spatial locality**, if the program often accesses memory locations with neighboring addresses at successive points in time during program execution. Thus, for programs with high spatial locality there is often the situation that after an access to a memory location, one or more memory locations of the same cache line are also accessed shortly afterward. In such situations, after loading a cache block, several of the following memory locations can be loaded from this cache block, thus avoiding expensive cache misses. The use of cache blocks comprising several memory words is based on the assumption that most programs exhibit spatial locality, i.e., when loading a cache block not only one but several memory words of the cache block are accessed before the cache block is replaced again.
- The memory accesses of a program have a high **temporal locality**, if it often happens that the *same* memory location is accessed multiple times at successive points in time during program execution. Thus, for programs with a high temporal

locality there is often the situation that after loading a cache block in the cache, the memory words of the cache block are accessed multiple times before the cache block is replaced again.

For programs with small spatial locality there is often the situation that after loading a cache block, only one of the memory words contained is accessed before the cache block is replaced again by another cache block. For programs with small temporal locality, there is often the situation that after loading a cache block because of a memory access, the corresponding memory location is accessed only once before the cache block is replaced again. Many program transformations to increase temporal or spatial locality of programs have been proposed, see [12, 175] for more details.

In the following, we give a short overview of important characteristics of caches. In particular, we consider cache size, mapping of memory blocks to cache blocks, replacement algorithms, and write-back policies. We also consider the use of multi-level caches.

2.7.1.1 Cache Size

Using the same hardware technology, the access time of a cache increases (slightly) with the size of the cache because of an increased complexity of the addressing. But using a larger cache leads to a smaller number of replacements as a smaller cache, since more cache blocks can be kept in the cache. The size of the caches is limited by the available chip area. Off-chip caches are rarely used to avoid the additional time penalty of off-chip accesses. Typical sizes for L1 caches lie between 8K and 128K memory words where a memory word is four or eight bytes long, depending on the architecture. During the last years, the typical size of L1 caches has not increased significantly.

If a cache miss occurs when accessing a memory location, an entire cache block is brought into the cache. For designing a memory hierarchy, the following points have to be taken into consideration when fixing the size of the cache blocks:

- Using larger blocks reduces the number of blocks that fit in the cache when using the same cache size. Therefore, cache blocks tend to be replaced earlier when using larger blocks compared to smaller blocks. This suggests to set the cache block size as small as possible.
- On the other hand, it is useful to use blocks with more than one memory word, since the transfer of a block with x memory words from main memory into the cache takes less time than x transfers of a single memory word. This suggests to use larger cache blocks.

As a compromise, a medium block size is used. Typical sizes for L1 cache blocks are four or eight memory words.

2.7.1.2 Mapping of Memory Blocks to Cache Blocks

Data is transferred between main memory and cache in blocks of a fixed length. Because the cache is significantly smaller than the main memory, not all memory blocks can be stored in the cache at the same time. Therefore, a mapping algorithm must be used to define at which position in the cache a memory block can be stored. The mapping algorithm used has a significant influence on the cache behavior and determines how a stored block is localized and retrieved from the cache. For the mapping, the notion of **associativity** plays an important role. Associativity determines at how many positions in the cache a memory block can be stored. The following methods are distinguished:

- for a **direct mapped** cache, each memory block can be stored at exactly one position in the cache;
- for a **fully associative** cache, each memory block can be stored at an arbitrary position in the cache;
- for a **set associative** cache, each memory block can be stored at a fixed number of positions.

In the following, we consider these three mapping methods in more detail for a memory system which consists of a main memory and a cache. We assume that the main memory comprises $n = 2^s$ blocks which we denote as B_j for $j = 0, \ldots, n-1$. Furthermore, we assume that there are $m = 2^r$ cache positions available; we denote the corresponding cache blocks as $\bar{B}_i$ for $i = 0, \ldots, m-1$. The memory blocks and the cache blocks have the same size of $l = 2^w$ memory words. At different points of program execution, a cache block may contain different memory blocks. Therefore, for each cache block a **tag** must be stored, which identifies the memory block that is currently stored. The use of this tag information depends on the specific mapping algorithm and will be described in the following. As running example, we consider a memory system with a cache of size 64 Kbytes which uses cache blocks of 4 bytes. Thus, 16K = 2^{14} blocks of four bytes each fit into the cache. With the notation from above, it is $r = 14$ and $w = 2$. The main memory is 4 Gbytes = 2^{32} bytes large, i.e., it is $s = 30$ if we assume that a memory word is one byte. We now consider the three mapping methods in turn.

2.7.1.3 Direct Mapped Caches

The simplest form to map memory blocks to cache blocks is implemented by **direct mapped** caches. Each memory block B_j can be stored at only one specific cache location. The mapping of a memory block B_j to a cache block $\bar{B}_i$ is defined as follows:

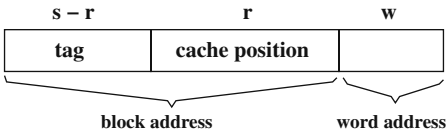
$$B_j \text{ is mapped to } \bar{B}_i \text{ if } i = j \bmod m.$$

Thus, there are $n/m = 2^{s-r}$ different memory blocks that can be stored in one specific cache block $\bar{B}_i$. Based on the mapping, memory blocks are assigned to cache positions as follows:

cache block	memory block
0	$0, m, 2m, \ldots, 2^s - m$
1	$1, m+1, 2m+1, \ldots, 2^s - m + 1$
$\vdots$	$\vdots$
$m-1$	$m-1, 2m-1, 3m-1, \ldots, 2^s - 1$

Since the cache size m is a power of 2, the modulo operation specified by the mapping function can be computed by using low-order bits of the memory address specified by the processor. Since a cache block contains $l = 2^w$ memory words, the memory address can be partitioned into a word address and a block address. The block address specifies the position of the corresponding memory block in main memory. It consists of the s most significant (leftmost) bits of the memory address. The word address specifies the position of the memory location in the memory block, relative to the first location of the memory block. It consists of the w least significant (rightmost) bits of the memory address.

For a direct mapped cache, the r rightmost bits of the block address of a memory location define at which of the $m = 2^r$ cache positions the corresponding memory block must be stored if the block is loaded into the cache. The remaining $s - r$ bits can be interpreted as tag which specifies which of the 2^{s-r} possible memory blocks is currently stored at a specific cache position. This tag must be stored with the cache block. Thus each memory address is partitioned as follows:



For the running example, the tags consist of $s - r = 16$ bits for a direct mapped cache.

Memory access is illustrated in Fig. 2.33(a) for an example memory system with block size 2 ($w = 1$), cache size 4 ($r = 2$), and main memory size 16 ($s = 4$). For each memory access specified by the processor, the cache position at which the requested memory block must be stored is identified by considering the r rightmost bits of the block address. Then the tag stored for this cache position is compared with the $s - r$ leftmost bits of the block address. If both tags are identical, the referenced memory block is currently stored in the cache, and memory access can be done via the cache. A *cache hit* occurs. If the two tags are different, the requested memory block must first be loaded into the cache at the given cache position before the memory location specified can be accessed.

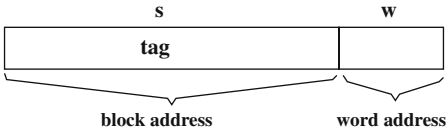
Direct mapped caches can be implemented in hardware without great effort, but they have the disadvantage that each memory block can be stored at only one cache position. Thus, it can happen that a program repeatedly specifies memory addresses in different memory blocks that are mapped to the same cache position. In this situation, the memory blocks will be continually loaded and replaced in the cache,

leading to a large number of cache misses and therefore a large execution time. This phenomenon is also called *thrashing*.

2.7.1.4 Fully Associative Caches

In a fully associative cache, each memory block can be placed in *any* cache position, thus overcoming the disadvantage of direct mapped caches. As for direct mapped caches, a memory address can again be partitioned into a block address (s leftmost bits) and a word address (w rightmost bits). Since each cache block can contain any memory block, the entire block address must be used as tag and must be stored with the cache block to allow the identification of the memory block stored. Thus, each memory address is partitioned as follows:

To check whether a given memory block is stored in the cache, all the entries in the cache must be searched, since the memory block can be stored at any cache position. This is illustrated in Fig. 2.33(b).

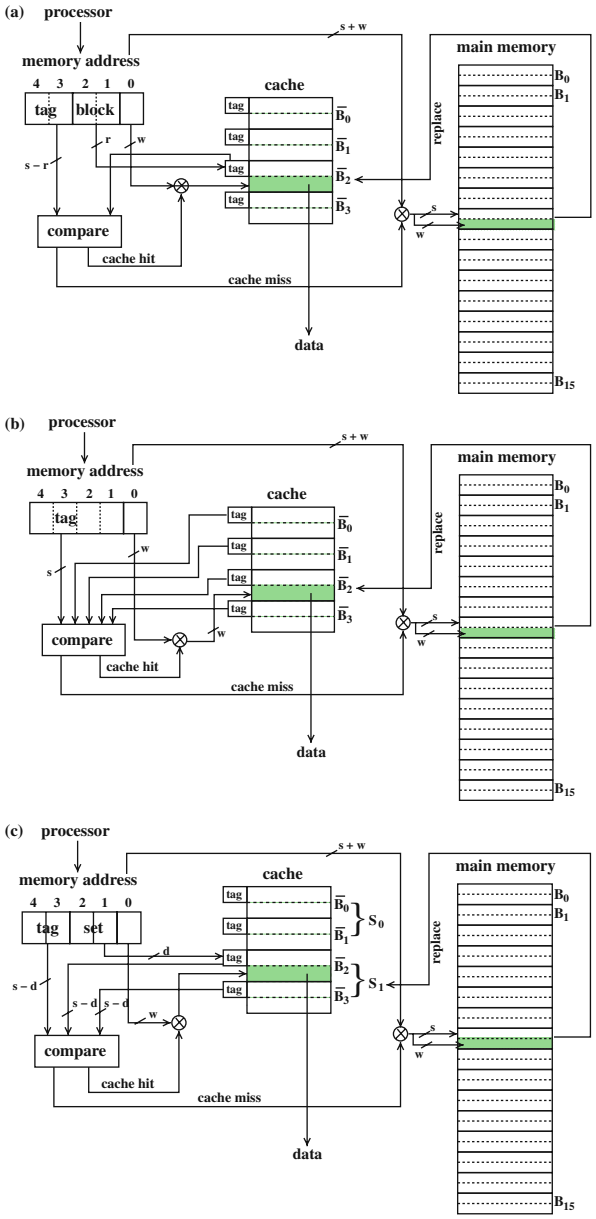


The advantage of fully associative caches lies in the increased flexibility when loading memory blocks into the cache. The main disadvantage is that for each memory access all cache positions must be considered to check whether the corresponding memory block is currently held in the cache. To make this search practical, it must be done in parallel using a separate comparator for each cache position, thus increasing the required hardware effort significantly. Another disadvantage is that the tags to be stored for each cache block are significantly larger as for direct mapped caches. For the example cache introduced above, the tags must be 30 bits long for a fully associated cache, i.e., for each 32-bit memory block, a 30-bit tag must be stored. Because of the large search effort, a fully associative mapping is useful only for caches with a small number of positions.

2.7.1.5 Set Associative Caches

Set associative caches are a compromise between direct mapped and fully associative caches. In a set associative cache, the cache is partitioned into v sets $S_0, \ldots, S_{v-1}$ where each set consists of $k = m/v$ blocks. A memory block B_j is not mapped to an individual cache block, but to a unique set in the cache. Within the set, the memory block can be placed in any cache block of that set, i.e., there are k different cache blocks in which a memory block can be stored. The set of a memory block B_j is defined as follows:

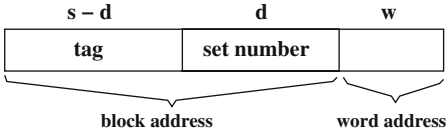
$$B_j \text{ is mapped to set } S_i, \text{ if } i = j \bmod v$$



This figure will be printed in b/w

Fig. 2.33 Illustration of the mapping of memory blocks to cache blocks for a cache with $m = 4$ cache blocks ($r = 2$) and a main memory with $n = 16$ memory blocks ($s = 4$). Each block contains two memory words ($w = 1$). (**a**) Direct mapped cache; (**b**) fully associative cache; (**c**) set associative cache with $k = 2$ blocks per set, using $v = 2$ sets ($d = 1$)

for $j = 0, \ldots, n-1$. A memory access is illustrated in Fig. 2.33(c). Again, a memory address consists of a block address (s bits) and a word address (w bits). The $d = \log v$ rightmost bits of the block address determine the set S_i to which the corresponding memory block is mapped. The leftmost $s - d$ bits of the block address are the tag that is used for the identification of the memory blocks stored in the individual cache blocks of a set. Thus, each memory address is partitioned as follows:



When a memory access occurs, the hardware first determines the set to which the memory block is assigned. Then, the tag of the memory block is compared with the tags of all cache blocks in the set. If there is a match, the memory access can be performed via the cache. Otherwise, the corresponding memory block must first be loaded into one of the cache blocks of the set.

For $v = m$ and $k = 1$, a set associative cache reduces to a direct mapped cache. For $v = 1$ and $k = m$, a fully associative cache results. Typical cases are $v = m/4$ and $k = 4$, leading to a *4-way set associative cache*, and $v = m/8$ and $k = 8$, leading to an *8-way set associative cache*. For the example cache, using $k = 4$ leads to 4K sets; $d = 12$ bits of the block address determine the set to which a memory block is mapped. The tags used for the identification of memory blocks within a set are 18 bits long.

2.7.1.6 Block Replacement Methods

When a cache miss occurs, a new memory block must be loaded into the cache. To do this for a fully occupied cache, one of the memory blocks in the cache must be replaced. For a direct mapped cache, there is only one position at which the new memory block can be stored, and the memory block occupying that position must be replaced. For a fully associative or set associative cache, there are several positions at which the new memory block can be stored. The block to be replaced is selected using a *replacement method*. A popular replacement method is **least recently used (LRU)** which replaces the block in a set that has not been used for the longest time.

For the implementation of the LRU method, the hardware must keep track for each block of a set when the block was used last. The corresponding time entry must be updated at each usage time of the block. This implementation requires additional space to store the time entries for each block and additional control logic to update the time entries. For a 2-way set associative cache the LRU method can be implemented more easily by keeping a USE bit for each of the two blocks in a set. When a cache block of a set is accessed, its USE bit is set to 1 and the USE bit of the other block in the set is set to 0. This is performed for each memory access. Thus,

the block whose USE bit is 1 has been accessed last, and the other block should be replaced if a new block has to be loaded into the set. An alternative to LRU is *least frequently used* (LFU) which replaces the block of a set that has experienced the fewest references. But the LFU method also requires additional control logic since for each block a counter must be maintained which must be updated for each memory access. For a larger associativity, an exact implementation of LRU or LFU as described above is often considered as too costly [84], and approximations or other schemes are used. Often, the block to be replaced is selected *randomly*, since this can be implemented easily. Moreover, simulations have shown that random replacement leads to only slightly inferior performance compared to more sophisticated methods like LRU or LFU [84, 164].

2.7.2 Write Policy

A cache contains a subset of the memory blocks. When the processor issues a *write access* to a memory block that is currently stored in the cache, the referenced block is definitely updated in the cache, since the next read access must return the most recent value. There remains the question: When is the corresponding memory block in the main memory updated? The earliest possible update time for the main memory is immediately after the update in the cache; the latest possible update time for the main memory is when the cache block is replaced by another block. The exact replacement time and update method is captured by the write policy. The most popular policies are **write-through** and **write-back**.

2.7.2.1 Write-Through Policy

Using write-through, a modification of a block in the cache using a write access is immediately transferred to main memory, thus keeping the cache and the main memory consistent. An advantage of this approach is that other devices like I/O modules that have direct access to main memory always get the newest value of a memory block. This is also important for multicore systems, since after a write by one processor, all other processors always get the most recently written value when accessing the same block. A drawback of write-through is that every write in the cache causes also a write to main memory which typically takes at least 100 processor cycles to complete. This could slow down the processor if it had to wait for the completion. To avoid processor waiting, a *write buffer* can be used to store pending write operations into the main memory [137, 84]. After writing the data into the cache and into the write buffer, the processor can continue its execution without waiting for the completion of the write into the main memory. A write buffer entry can be freed after the write into main memory completes. When the processor performs a write and the write buffer is full, a write stall occurs, and the processor must wait until there is a free entry in the write buffer.

2.7.2.2 Write-Back Policy

Using write-back, a write operation to a memory block that is currently held in the cache is performed only in the cache; the corresponding memory entry is not updated immediately. Thus, the cache may contain newer values than the main memory. The modified memory block is written to the main memory when the cache block is replaced by another memory block. To check whether a write to main memory is necessary when a cache block is replaced, a separate bit (*dirty bit*) is held for each cache block which indicates whether the cache block has been modified or not. The dirty bit is initialized to 0 when a block is loaded into the cache. A write access to a cache block sets the dirty bit to 1, indicating that a write to main memory must be performed when the cache block is replaced.

Using write-back policy usually leads to fewer write operations to main memory than write-through policy, since cache blocks can be written multiple times before they are written back to main memory. The drawback of write-back is that the main memory may contain invalid entries, and hence I/O modules can access main memory only through the cache.

If a write to a memory location goes to a memory block that is currently not in the cache, most caches use the *write-allocate* method: The corresponding memory block is first brought into the cache and then the modification is performed as described above. An alternative approach is *write no allocate*, which modifies in main memory without loading it into the cache. However, this approach is used less often.

2.7.2.3 Number of Caches

So far, we have considered the behavior of a single cache which is placed between the processor and main memory and which stores data blocks of a program in execution. Such caches are also called **data caches**.

Besides the program data, a processor also accesses instructions of the program in execution before they are decoded and executed. Because of loops in the program, an instruction can be accessed multiple times. To avoid multiple loading operations from main memory, instructions are also held in cache. To store instructions and data, a single cache can be used (*unified cache*). But often, two separate caches are used on the first level, an **instruction cache** to store instructions and a separate data cache to store data. This approach is also called *split caches*. This enables a greater flexibility for the cache design, since the data and instruction caches can work independently of each other and may have different size and associativity depending on the specific needs.

In practice, multiple levels of caches are typically used as illustrated in Fig. 2.34. The current standard is to have two levels with a trend toward three levels. For the first level (L1), split caches are typically used; for the remaining levels, unified caches are standard. The caches are hierarchically organized, and for two levels, the L1 caches contain a subset of the L2 cache which contains a subset of the main memory.

The caches are normally integrated into the chip area of the processor. Typical cache sizes lie between 8 Kbytes and 128 Kbytes for the L1 cache and between

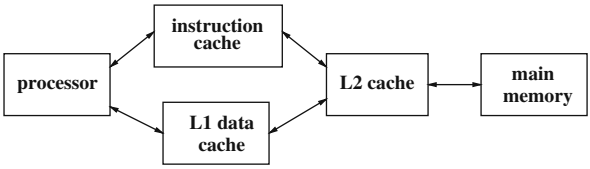


Fig. 2.34 Illustration of a two-level cache hierarchy

256 Kbytes and 8 Mbytes for the L2 cache. Typical sizes of the main memory lie between 1 Gbyte and 16 Gbytes. Typical access times are one or a few processor cycles for the L1 cache, between 15 and 25 cycles for the L2 cache, between 100 and 1000 cycles for the main memory, and between 10 and 100 million cycles for the hard disc [137].

2.7.3 Cache Coherency

Using a memory hierarchy with multiple levels of caches can help to bridge large access times to main memory. But the use of caches introduces the effect that memory blocks can be held in multiple copies in caches and main memory, and after an update in the L1 cache, other copies might become invalid, in particular if a write-back policy is used. This does not cause a problem as long as a single processor is the only accessing device. But if there are multiple accessing devices, as is the case for multicore processors, inconsistent copies can occur and should be avoided, and each execution core should always access the most recent value of a memory location. The problem of keeping the different copies of a memory location consistent is also referred to as *cache coherency problem*.

In a multiprocessor system with different cores or processors, in which each processor has a separate local cache, the same memory block can be held as copy in the local cache of multiple processors. If one or more of the processors update a copy of a memory block in their local cache, the other copies become invalid and contain inconsistent values. The problem can be illustrated for a bus-based system with three processors [35] as shown in the following example.

Example We consider a bus-based SMP system with three processors P_1, P_2, P_3 where each processor P_i has a local cache C_i for $i = 1, 2, 3$. The processors are connected to a shared memory M via a central bus. The caches C_i use a write-through strategy. We consider a variable u with initial value 5 which is held in the main memory before the following operations are performed at times t_1, t_2, t_3, t_4:

- t_1: Processor P_1 reads variable u. The memory block containing u is loaded into cache C_1 of P_1.
- t_2: Processor P_3 reads variable u. The memory block containing u is also loaded into cache C_3 of P_3.
- t_3: Processor P_3 writes the value 7 into u. This new value is also written into the main memory because write-through is used.
- t_4: Processor P_1 reads u by accessing the copy in its local cache.

At time t_4, processor P_1 reads the old value 5 instead of the new value 7, i.e., a cache coherency problem occurs. This is the case for both write-through and write-back caches: For write-through caches, at time t_3 the new value 7 is directly written into the main memory by processor P_3, but the cache of P_1 will not be updated. For write-back caches, the new value of 7 is not even updated in main memory, i.e., if another processor P_2 reads the value of u after time t_3, it will obtain the old value, even when the variable u is not held in the local cache of P_2.

For a correct execution of a parallel program on a shared address space, it must be ensured that for each possible order of read and write accesses performed by the participating processors according to their program statements, each processor obtains the right value, no matter whether the corresponding variable is held in cache or not.

The behavior of a memory system for read and write accesses performed by *different* processors to the *same* memory location is captured by the **coherency of the memory system**. Informally, a memory system is coherent if for each memory location any read access returns the most recently written value of that memory location. Since multiple processors may perform write operations to the same memory location at the same time, we must first define more precisely what the most recently written value is. For this definition, the order of the memory accesses in the parallel program executed is used as time measure, not the physical point in time at which the memory accesses are executed by the processors. This makes the definition independent of the specific execution environment and situation.

Using the program order of memory accesses, a memory system is coherent, if the following conditions are fulfilled [84]:

1. If a processor P writes into a memory location x at time t_1 and reads from the same memory location x at time $t_2 > t_1$ and if between t_1 and t_2 no other processor performs a write into x, then P obtains at time t_2 the value written by itself at time t_1. Thus, for each processor the order of the memory accesses in its program is preserved despite a parallel execution.
2. If a processor P_1 writes into a memory location x at time t_1 and if another processor P_2 reads x at time $t_2 > t_1$, then P_2 obtains the value written by P_1, if between t_1 and t_2 no other processors write into x and if the period of time $t_2 - t_1$ is sufficiently large. Thus, a value written by one of the processors must become visible to the other processors after a certain amount of time.
3. If two processors write into the same memory location x, these write operations are *serialized* so that all processors see the write operations in the *same order*. Thus, a global **write serialization** is performed.

To be coherent, a memory system must fulfill these three properties. In particular, for a memory system with caches which can store multiple copies of memory blocks, it must be ensured that each processor has a coherent view of the memory system through its local caches. To ensure this, hardware-based *cache coherence protocols* are used. Depending on the architecture of the execution platform, different protocols are used, including snooping protocols and directory-based protocols.

2.7.3.1 Snooping Protocols

The technique of bus snooping has first been used for bus-based SMP systems, where the local caches of the processors use a write-through policy. The technique relies on the property that on such systems all memory accesses are performed via the central bus, i.e., the bus is used as broadcast medium. Thus, all memory accesses can be observed by the cache controllers of all processors. Each cache controller can observe the memory accesses transferred over the bus. When the cache controller observes a write into a memory location that is currently held in the local cache, it updates the value in the cache by copying the new value from the bus into the cache. Thus, the local caches always contain the most recently written values of memory locations. These protocols are also called *update-based protocols*, since the cache controllers directly perform an update. There are also *invalidation-based protocols* in which the cache block corresponding to a memory block is invalidated so that the next read access must perform an update from main memory first. Using an update-based protocol in the example from above (p. 75), processor P_1 can observe the write operation of P_3 at time t_3 and can update the value of u in its local cache C_1 accordingly. Thus, at time t_4, P_1 reads the correct value 7.

The technique of bus snooping relies on the use of a write-through policy and the existence of a broadcast medium so that each cache controller can observe all write accesses to perform updates or invalidations. In the past, the broadcast medium has been a shared bus, but for newer architectures interconnection networks like crossbars or point-to-point networks are used. This makes updates or invalidations more complicated, since the interprocessor links are not shared, and the coherency protocol must use broadcasts to find potentially shared copies of memory blocks, see [84] for more details. Due to the coherence protocol, additional traffic occurs in the interconnection network, which may limit the effective memory access time of the processors. Snooping protocols are not restricted to write-through caches. The technique can also be applied to write-back caches as described in the following.

2.7.3.2 Write-Back Invalidation Protocol

In the following, we describe a basic write-back invalidation protocol, see [35, 84] for more details. In the protocol, each cache block can be in one of three states [35]:

M (modified) means that the cache block contains the current value of the memory block and that all other copies of this memory block in other caches or in the main memory are invalid, i.e., the block has been updated in the cache.

S (shared) means that the cache block has not been updated in this cache and that this cache contains the current value, as do the main memory and zero or more other caches.

I (invalid) means that the cache block does not contain the most recent value of the memory block.

According to these three states, the protocol is also called **MSI protocol**. The same memory block can be in different states in different caches. Before a processor

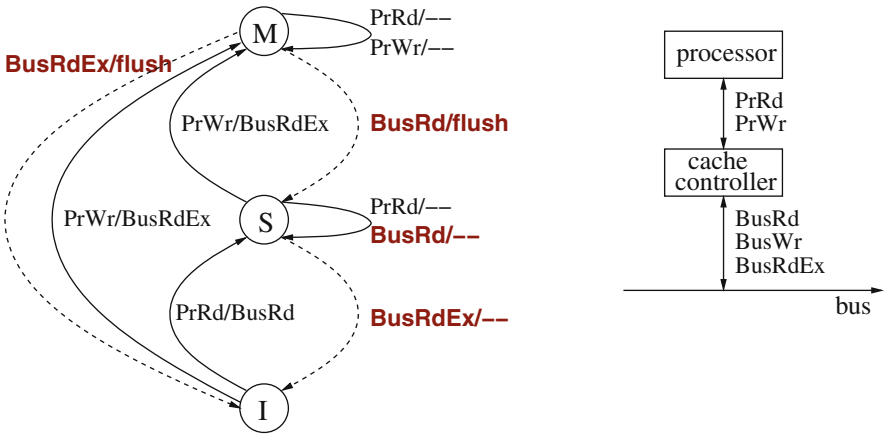
modifies a memory block in its local cache, all other copies of the memory block in other caches and the main memory are marked as invalid (I). This is performed by an operation on the broadcast medium. After that, the processor can perform one or several write operations to this memory block without performing other invalidations. The memory block is marked as modified (M) in the cache of the writing processor. The protocol provides three operations on the broadcast medium, which is a shared bus in the simplest case:

- **Bus Read** (`BusRd`): This operation is generated by a read operation (`PrRd`) of a processor to a memory block that is currently not stored in the cache of this processor. The cache controller requests a copy of the memory block by specifying the corresponding memory address. The requesting processor does not intend to modify the memory block. The most recent value of the memory block is provided from the main memory or from another cache.
- **Bus Read Exclusive** (`BusRdEx`): This operation is generated by a write operation (`PrWr`) of a processor to a memory block that is currently not stored in the cache of this processor or that is currently not in the M state in this cache. The cache controller requests an exclusive copy of the memory block that it intends to modify; the request specifies the corresponding memory address. The memory system provides the most recent value of the memory block. All other copies of this memory block in other caches are marked invalid (I).
- **Write-Back** (`BusWr`): The cache controller writes a cache block that is marked as modified (M) back to the main memory. This operation is generated if the cache block is replaced by another memory block. After the operation, the main memory contains the latest value of the memory block.

The processor performs the usual read and write operations (`PrRd`, `PrWr`) to memory locations, see Fig. 2.35 (right). The cache controller provides the requested memory words to the processor by loading them from the local cache. In case of a cache miss, this includes the loading of the corresponding memory block using a bus operation. The exact behavior of the cache controller depends on the state of the cache block addressed and can be described by a state transition diagram that is shown in Fig. 2.35 (left).

A read and write operation to a cache block marked with M can be performed in the local cache without a bus operation. The same is true for a read operation to a cache block that is marked with S. To perform a write operation to a cache block marked with S, the cache controller must first execute a `BusRdEx` operation to become the exclusive owner of the cache block. The local state of the cache block is transformed from S to M. The cache controllers of other processors that have a local copy of the same cache block with state S observe the `BusRdEx` operation and perform a local state transition from S to I for this cache block.

When a processor tries to read a memory block that is not stored in its local cache or that is marked with I in its local cache, the corresponding cache controller performs a `BusRd` operation. This causes a valid copy to be stored in the local cache marked with S. If another processor observes a `BusRd` operation for a memory



This figure will be printed in b/w

Fig. 2.35 Illustration of the MSI protocol: Each cache block can be in one of the states M (modified), S (shared), or I (invalid). State transitions are shown by *arcs* that are annotated with operations. A state transition can be caused by
(a) Operations of the processor (`PrRd`, `PrWr`) (*solid arcs*); The bus operations initiated by the cache controller are annotated behind the slash sign. If no bus operation is shown, the cache controller only accesses the local cache.
(b) Operations on the bus observed by the cache controller and issued by the cache controller of other processors (*dashed arcs*). Again, the corresponding operations of the local cache controller are shown behind the slash sign. The operation *flush* means that the cache controller puts the value of the requested memory block on the bus, thus making it available to other processors. If no arc is shown for a specific bus operation observed for a specific state, no state transition occurs and the cache controller does not need to perform an operation

block, for which it has the only valid copy (state M), it puts the value of the memory block on the bus and marks its local copy with state S (shared).

When a processor tries to write into a memory block that is not stored in its local cache or that is marked with I, the cache controller performs a `BusRdEx` operation. This provides a valid copy of the memory block in the local cache, which is marked with M, i.e., the processor is the exclusive owner of this memory block. If another processor observes a `BusRdEx` operation for a memory block which is marked with M in its local cache, it puts the value of the memory block on the bus and performs a local state transition from M to I.

A drawback of the MSI protocol is that a processor which first reads a memory location and then writes into a memory location must perform two bus operations `BusRd` and `BusRdEx`, even if no other processor is involved. The `BusRd` provides the memory block in S state, the `BusRdEx` causes a state transition from S to M. This drawback can be eliminated by adding a new state E (exclusive):

E (exclusive) means that the cache contains the only (exclusive) copy of the memory block and that this copy has not been modified. The main memory contains a valid copy of the block, but no other processor is caching this block.

If a processor requests a memory block by issuing a `PrRd` and if no other processor has a copy of this memory block in its local cache, then the block is marked with E (instead of S in the MSI protocol) in the local cache after being loaded from the main memory with a `BusRd` operation. If at a later time, this processor performs a write into this memory block, a state transition from E to M is performed before the write. In this case, no additional bus operation is necessary. If between the local read and write operation, another processor performs a read to the same memory block, the local state is changed from E to S. The local write would then cause the same actions as in the MSI protocol. The resulting protocol is called **MESI protocol** according to the abbreviation of the four states. A more detailed discussion and a detailed description of several variants can be found in [35]. Variants of the MESI protocol are supported by many processors and the protocols play an important role in multicore processors to ensure the coherency of the local caches of the cores.

The MSI and MESI protocols are invalidation protocols. An alternative is **write-back update protocols** for write-back caches. In these protocols, after an update of a cache block with state M, all other caches which also contain a copy of the corresponding memory block are updated. Therefore, the local caches always contain the most recent values of the cache blocks. In practice, these protocols are rarely used because they cause more traffic on the bus.

2.7.3.3 Directory-Based Cache Coherence Protocols

Snooping protocols rely on the existence of a shared broadcast medium like a bus or a switch through which all memory accesses are transferred. This is typically the case for multicore processors or small SMP systems. But for larger systems, such a shared medium often does not exist and other mechanisms have to be used.

A simple solution would be not to support cache coherence at hardware level. Using this approach, the local caches would only store memory blocks of the local main memory. There would be no hardware support to store memory blocks from the memory of other processors in the local cache. Instead, software support could be provided, but this requires more support from the programmer and is typically not as fast as a hardware solution.

An alternative to snooping protocols are **directory-based protocols**. These do not rely on a shared broadcast medium. Instead, a central directory is used to store the state of every memory block that may be held in cache. Instead of observing a shared broadcast medium, a cache controller can get the state of a memory block by a lookup in the directory. The directory can be held shared, but it could also be distributed among different processors to avoid bottlenecks when the directory is accessed by many processors. In the following, we give a short overview of directory-based protocols. For a more detailed description, we refer again to [35, 84].

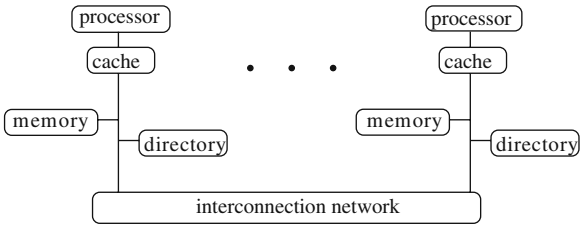


Fig. 2.36 Directory-based cache coherency

As example, we consider a parallel machine with a distributed memory. We assume that for each local memory a directory is maintained that specifies for each memory block of the local memory which caches of other processors currently store a copy of this memory block. For a parallel machine with p processors the directory can be implemented by maintaining a bit vector with p *presence bits* and a number of state bits for each memory block. Each presence bit indicates whether a specific processor has a valid copy of this memory block in its local cache (value 1) or not (value 0). An additional *dirty bit* is used to indicate whether the local memory contains a valid copy of the memory block (value 0) or not (value 1). Each directory is maintained by a *directory controller* which updates the directory entries according to the requests observed on the network.

Figure 2.36 illustrates the organization. In the local caches, the memory blocks are marked with M (modified), S (shared), or I (invalid), depending on their state, similar to the snooping protocols described above. The processors access the memory system via their local cache controllers. We assume a global address space, i.e., each memory block has a memory address which is unique in the entire parallel system.

When a read miss or write miss occurs at a processor i, the associated cache controller contacts the local directory controller to obtain information about the accessed memory block. If this memory block belongs to the local memory and the local memory contains a valid copy (dirty bit 0), the memory block can be loaded into the cache with a local memory access. Otherwise, a non-local (remote) access must be performed. A request is sent via the network to the directory controller at the processor owning the memory block (home node). For a read miss, the receiving directory controller reacts as follows:

- If the dirty bit of the requested memory block is 0, the directory controller retrieves the memory block from local memory and sends it to the requesting node via the network. The presence bit of the receiving processor i is set to 1 to indicate that i has a valid copy of the memory block.
- If the dirty bit of the requested memory block is 1, there is exactly one processor j which has a valid copy of the memory block; the presence bit of this processor is 1. The directory controller sends a corresponding request to this processor j. The cache controller of j sets the local state of the memory block from M to S and sends the memory block both to the home node of the memory block and to the processor i from which the original request came. The directory controller of

the home node stores the current value in the local memory, sets the dirty bit of the memory block to 0, and sets the presence bit of processor i to 1. The presence bit of j remains 1.

For a write miss, the receiving directory controller does the following:

- If the dirty bit of the requested memory block is 0, the local memory of the home node contains a valid copy. The directory controller sends an invalidation request to all processors j for which the presence bit is 1. The cache controllers of these processors set the state of the memory block to I. The directory controller waits for an acknowledgment from these cache controllers, sets the presence bit for these processors to 0, and sends the memory block to the requesting processor i. The presence bit of i is set to 1, the dirty bit is also set to 1. After having received the memory block, the cache controller of i stores the block in its cache and sets its state to M.
- If the dirty bit of the requested memory block is 1, the memory block is requested from the processor j whose presence bit is 1. Upon arrival, the memory block is forwarded to processor i, the presence bit of i is set to 1, and the presence bit of j is set to 0. The dirty bit remains at 1. The cache controller of j sets the state of the memory block to I.

When a memory block with state M should be replaced by another memory block in the cache of processor i, it must be written back into its home memory, since this is the only valid copy of this memory block. To do so, the cache controller of i sends the memory block to the directory controller of the home node. This one writes the memory block back to the local memory and sets the dirty bit of the block and the presence bit of processor i to 0.

A cache block with state S can be replaced in a local cache without sending a notification to the responsible directory controller. Sending a notification avoids the responsible directory controller sending an unnecessary invalidation message to the replacing processor in case of a write miss as described above.

The directory protocol just described is kept quite simple. Directory protocols used in practice are typically more complex and contain additional optimizations to reduce the overhead as far as possible. Directory protocols are typically used for distributed memory machines as described. But they can also be used for shared memory machines. Examples are the Sun T1 and T2 processors, see [84] for more details.

2.7.4 Memory Consistency

Cache coherence ensures that each processor of a parallel system has the same consistent view of the memory through its local cache. Thus, at each point in time, each processor gets the same value for each variable if it performs a read access. But cache coherence does not specify in which order write accesses become visible to the other processors. This issue is addressed by memory consistency models. These

models provide a formal specification of how the memory system will appear to the programmer. The consistency model sets some restrictions on the values that can be returned by a read operation in a shared address space. Intuitively, a read operation should always return the value that has been written last. In uniprocessors, the program order uniquely defines which value this is. In multiprocessors, different processors execute their programs concurrently and the memory accesses may take place in different order depending on the relative progress of the processors.

The following example illustrates the different results of a parallel program if different execution orders of the program statements by the different processors are considered, see also [95].

Example We consider three processors P_1, P_2, P_3 which execute a parallel program with shared variables x_1, x_2, x_3. The three variables x_1, x_2, x_3 are assumed to be initialized to 0. The processors execute the following programs:

processor	P_1	P_2	P_3
program	(1) $x_1 = 1$;	(3) $x_2 = 1$;	(5) $x_3 = 1$;
	(2) print x_2, x_3;	(4) print x_1, x_3;	(6) print x_1, x_2;

Processor P_i sets the value of $x_i, i = 1, 2, 3$, to 1 and prints the values of the other variables x_j for $j \neq i$. In total, six values are printed which may be 0 or 1. Since there are no dependencies between the two statements executed by P_1, P_2, P_3, their order can be arbitrarily reversed. If we allow such a reordering and if the statements of the different processors can be mixed arbitrarily, there are in total $2^6 = 64$ possible output combinations consisting of 0 and 1. Different global orders may lead to the same output. If the processors are restricted to execute their statements in program order (e.g., P_1 must execute (1) before (2)), then output 000000 is *not* possible, since at least one of the variables x_1, x_2, x_3 must be set to 1 before a print operation occurs. A possible sequentialization of the statements is (1), (2), (3), (4), (5), (6). The corresponding output is 001011.

To clearly describe the behavior of the memory system in multiprocessor environments, the concept of consistency models has been introduced. Using a consistency model, there is a clear definition of the allowable behavior of the memory system which can be used by the programmer for the design of parallel programs. The situation can be described as follows [165]: The input to the memory system is a set of memory accesses (read or write) which are partially ordered by the program order of the executing processors. The output of the memory system is a collection of values returned by the read accesses executed. A consistency model can be seen as a function that maps each input to a set of allowable outputs. The memory system using a specific consistency model guarantees that for any input, only outputs from the set of allowable outputs are produced. The programmer must write parallel programs such that they work correctly for any output allowed by the consistency model. The use of a consistency model also has the advantage that it abstracts from the specific physical implementation of a memory system and provides a clear abstract interface for the programmer.

In the following, we give a short overview of popular consistency models. For a more detailed description, we refer to [3, 35, 84, 111, 165].

Memory consistency models can be classified according to the following two criteria:

- Are the memory access operations of each processor executed in program order?
- Do all processors observe the memory access operations performed in the same order?

Depending on the answer to these questions, different consistency models can be identified.

2.7.4.1 Sequential Consistency

A popular model for memory consistency is the *sequential consistency model* (SC model) [111]. This model is an intuitive extension of the uniprocessor model and places strong restrictions on the execution order of the memory accesses. A memory system is sequentially consistent, if the memory accesses of each single processor are performed in the program order described by that processor's program and if the global result of all memory accesses of all processors appears to all processors in the *same* sequential order which results from an arbitrary interleaving of the memory accesses of the different processors. Memory accesses must be performed as *atomic* operations, i.e., the effect of each memory operation must become globally visible to all processors before the next memory operation of any processor is started.

The notion of program order leaves some room for interpretation. Program order could be the order of the statements performing memory accesses in the *source program*, but it could also be the order of the memory access operations in a machine program generated by an optimizing compiler which could perform statement reordering to obtain a better performance. In the following, we assume that the order in the source program is used.

Using sequential consistency, the memory operations are treated as atomic operations that are executed in the order given by the source program of each processor and that are centrally sequentialized. This leads to a *total order* of the memory operations of a parallel program which is the same for all processors of the system. In the example given above, not only output 001011 but also 111111 conforms to the SC model. The output 011001 is not possible for sequential consistency.

The requirement of a total order of the memory operations is a stronger restriction as has been used for the coherence of a memory system in the last section (p. 76). For a memory system to be coherent it is required that the write operations to the *same* memory location are sequentialized such that they appear to all processors in the same order. But there is no restriction on the order of write operations to different memory locations. On the other hand, sequential consistency requires that all write operations (to arbitrary memory locations) appear to all processors in the same order.

The following example illustrates that the atomicity of the write operations is important for the definition of sequential consistency and that the requirement of a sequentialization of the write operations alone is not sufficient.

Example Three processors P_1, P_2, P_3 execute the following statements:

processor	P_1	P_2	P_3
program	(1) $x_1 = 1$;	(2) while($x_1 == 0$);	(4) while($x_2 == 0$);
		(3) $x_2 = 1$;	(5) print(x_1);

The variables x_1 and x_2 are initialized to 0. Processor P_2 waits until x_1 has value 1 and then sets x_2 to 1. Processor P_3 waits until x_2 has value 1 and then prints the value of x_1. Assuming atomicity of write operations, the statements are executed in the order (1), (2), (3), (4), (5), and processor P_3 prints the value 1 for x_1, since write operation (1) of P_1 must become visible to P_3 before P_2 executes write operation (3). Using a sequentialization of the write operations of a variable without requiring atomicity and global sequentialization as is required for sequential consistency would allow the execution of statement (3) before the effect of (1) becomes visible to P_3. Thus, (5) could print the value 0 for x_1.

To further illustrate this behavior, we consider a directory-based protocol and assume that the processors are connected via a network. In particular, we consider a directory-based invalidation protocol to keep the caches of the processors coherent. We assume that the variables x_1 and x_2 have been initialized to 0 and that they are both stored in the local caches of P_2 and P_3. The cache blocks are marked as shared (S).

The operations of each processor are executed in program order and a memory operation is started not before the preceding operations of the same processor have been completed. Since no assumptions on the transfer of the invalidation messages in the network are made, the following execution order is possible:

(1) P_1 executes the write operation (1) to x_1. Since x_1 is not stored in the cache of P_1, a write miss occurs. The directory entry of x_1 is accessed and invalidation messages are sent to P_2 and P_3.

(2) P_2 executes the read operation (2) to x_1. We assume that the invalidation message of P_1 has already reached P_2 and that the memory block of x_1 has been marked invalid (I) in the cache of P_2. Thus, a read miss occurs, and P_2 obtains the current value 1 of x_1 over the network from P_1. The copy of x_1 in the main memory is also updated.

After having received the current value of x_1, P_1 leaves the while loop and executes the write operation (3) to x_2. Because the corresponding cache block is marked as shared (S) in the cache of P_2, a write miss occurs. The directory entry of x_2 is accessed and invalidation messages are sent to P_1 and P_3.

(3) P_3 executes the read operation (4) to x_2. We assume that the invalidation message of P_2 has already reached P_3. Thus, P_3 obtains the current value 1 of x_2 over the network. After that, P_3 leaves the while loop and executes the print operation (5). Assuming that the invalidation message of P_1 for x_1 has not yet reached P_3, P_3 accesses the old value 0 for x_1 from its local cache, since the

corresponding cache block is still marked with S. This behavior is possible if the invalidation messages have different transfer times over the network.

In this example, sequential consistency is violated, since the processors observe different orders of the write operation: Processor P_2 observes the order $x_1 = 1, x_2 = 1$ whereas P_3 observes the order $x_2 = 1, x_1 = 1$ (since P_3 gets the *new* value of x_2, but the *old* value of x_1 for its read accesses).

In a parallel system, sequential consistency can be guaranteed by the following *sufficient conditions* [35, 45, 157]:

(1) Every processor issues its memory operations in program order. In particular, the compiler is not allowed to change the order of memory operations, and no out-of-order executions of memory operations are allowed.
(2) After a processor has issued a write operation, it waits until the write operation has been completed before it issues the next operation. This includes that for a write miss all cache blocks which contain the memory location written must be marked invalid (I) before the next memory operation starts.
(3) After a processor has issued a read operation, it waits until this read operation and the write operation whose value is returned by the read operation have been entirely completed. This includes that the value returned to the issuing processor becomes visible to all other processors before the issuing processor submits the next memory operation.

These conditions do not contain specific requirements concerning the interconnection network, the memory organization, or the cooperation of the processors in the parallel system. In the example from above, condition (3) ensures that after reading x_1, P_2 waits until the write operation (1) has been completed before it issues the next memory operation (3). Thus, P_3 always reads the new value of x_1 when it reaches statement (5). Therefore, sequential consistency is ensured.

For the programmer, sequential consistency provides an easy and intuitive model. But the model has a performance disadvantage, since all memory accesses must be atomic and since memory accesses must be performed one after another. Therefore, processors may have to wait for quite a long time before memory accesses that they have issued have been completed. To improve performance, consistency models with fewer restrictions have been proposed. We give a short overview in the following and refer to [35, 84] for a more detailed description. The goal of the less restricted models is to still provide a simple and intuitive model but to enable a more efficient implementation.

2.7.4.2 Relaxed Consistency Models

Sequential consistency requires that the read and write operations issued by a processor maintain the following orderings where $X \rightarrow Y$ means that the operation X must be completed before operation Y is executed:

- $R \rightarrow R$: The read accesses are performed in program order.
- $R \rightarrow W$: A read operation followed by a write operation is executed in program order. If both operations access the same memory location, an *anti-dependence* occurs. In this case, the given order must be preserved to ensure that the read operation accesses the correct value.
- $W \rightarrow W$: The write accesses are performed in program order. If both operations access the same memory location, an *output dependence* occurs. In this case, the given order must be preserved to ensure that the correct value is written last.
- $W \rightarrow R$: A write operation followed by a read operation is executed in program order. If both operations access the same memory location, a *flow dependence* (also called *true dependence*) occurs.

If there is a dependence between the read and write operations the given order must be preserved to ensure the correctness of the program. If there is no such dependence, the given order must be kept to ensure sequential consistency. *Relaxed consistency models* abandon one or several of the orderings required for sequential consistency, if the data dependencies allow this.

Processor consistency models relax the $W \rightarrow R$ ordering to be able to partially hide the latency of write operations. Using this relaxation, a processor can execute a read operation even if a preceding write operation has not yet been completed if there are no dependencies. Thus, a read operation can be performed even if the effect of a preceding write operation is not visible yet to all processors. Processor consistency models include *total store ordering* (**TSO model**) and *processor consistency* (**PC model**). In contrast to the TSO model, the PC model does not guarantee atomicity of the write operations. The differences between sequential consistency and the TSO or PC model are illustrated in the following example.

Example Two processors P_1 and P_2 execute the following statements:

processor	P_1	P_2
program	(1) $x_1 = 1$;	(3) $x_2 = 1$;
	(2) print(x_2);	(4) print(x_1);

Both variables x_1 and x_2 are initialized to 0. Using sequential consistency, statement (1) must be executed before statement (2), and statement (3) must be executed before statement (4). Thus, it is not possible that the value 0 is printed for both x_1 and x_2. But using TSO or PC, this output is possible, since, for example, the write operation (3) does not need to be completed before P_2 reads the value of x_1 in (4). Thus, both P_1 and P_2 may print the old value for x_1 and x_2, respectively.

Partial store ordering (PSO) models relax both the $W \rightarrow W$ and the $W \rightarrow R$ ordering required for sequential consistency. Thus in PSO models, write operations can be completed in a different order as given in the program if there is no output dependence between the write operations. Successive write operations can be overlapped which may lead to a faster execution, in particular when write misses occur. The following example illustrates the differences between the different models.

Example We assume that the variables x_1 and *flag* are initialized to 0. Two processors P_1 and P_2 execute the following statements:

processor	P_1	P_2
program	(1) $x_1 = 1$;	(3) while(flag == 0);
	(2) flag = 1;	(4) print(x_1);

Using sequential consistency, PC, or TSO, it is *not* possible that the value 0 is printed for x_1. But using the PSO model, the write operation (2) can be completed *before* $x_1 = 1$. Thus, it is possible that the value 0 is printed for x_1 in statement (4). This output does not conform to intuitive understanding of the program behavior in the example, making this model less attractive for the programmer.

Weak ordering models additionally relax the $R \rightarrow R$ and $R \rightarrow W$ orderings. Thus, no completion order of the memory operations is guaranteed. To support programming, these models provide additional synchronization operations to ensure the following properties:

- All read and write operations which lie in the program *before* the synchronization operation are completed before the synchronization operation.
- The synchronization operation is completed before read or write operations are started which lie in the program *after* the synchronization operation.

The advent of multicore processors has led to an increased availability of parallel systems and most processors provide hardware support for a memory consistency model. Often, relaxed consistency models are supported, as is the case for the PowerPC architecture of IBM or the different Intel architectures. But different hardware manufacturers favor different models, and there is no standardization as yet.

2.8 Exercises for Chap. 2

Exercise 2.1 Consider a two-dimensional mesh network with n rows and m columns. What is the bisection bandwidth of this network?

Exercise 2.2 Consider a shuffle–exchange network with $n = 2^k$ nodes, $k > 1$. How many of the $3 \cdot 2^{k-1}$ edges are shuffle edges and how many are exchange edges? Draw a shuffle–exchange network for $k = 4$.

Exercise 2.3 In Sect. 2.5.2, p. 35, we have shown that there exist k independent paths between any two nodes of a k-dimensional hypercube network. For $k = 5$, determine all paths between the following pairs of nodes: (i) nodes 01001 and 00011; (ii) nodes 00001 and 10000.

Exercise 2.4 Write a (sequential) program that determines all paths between any two nodes for hypercube networks of arbitrary dimension.

Exercise 2.5 The RGC sequences RGC_k can be used to compute embeddings of different networks into a hypercube network of dimension k. Determine RGC_3, RGC_4,

and RGC_5. Determine an embedding of a three-dimensional mesh with $4 \times 2 \times 4$ nodes into a five-dimensional hypercube network.

Exercise 2.6 Show how a complete binary tree with n leaves can be embedded into a butterfly network of dimension $\log n$. The leaves of the trees correspond to the butterfly nodes at level $\log n$.

Exercise 2.7 Construct an embedding of a three-dimensional torus network with $8 \times 8 \times 8$ nodes into a nine-dimensional hypercube network according to the construction in Sect. 2.5.3, p. 39.

Exercise 2.8 A k-dimensional Beneš network consists of two connected k-dimensional butterfly networks, leading to $2k + 1$ stages, see p. 45. A Beneš network is *non-blocking*, i.e., any permutation between input nodes and output nodes can be realized without blocking. Consider an 8×8 Beneš network and determine the switch positions for the following two permutations:

$$\pi_1 = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 0 & 1 & 2 & 4 & 3 & 5 & 7 & 6 \end{pmatrix}, \quad \pi_2 = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 2 & 7 & 4 & 6 & 0 & 5 & 3 & 1 \end{pmatrix}.$$

Exercise 2.9 The cross-product $G_3 = (V_3, E_3) = G_1 \otimes G_2$ of two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ can be defined as follows:
$V_3 = V_1 \times V_2$ and $E_3 = \{((u_1, u_2), (v_1, v_2)) \mid ((u_1 = v_1)$ and $(u_2, v_2) \in E_2)$ or $((u_2 = v_2)$ and $(u_1, v_1) \in E_1)\}$. The symbol $\bigotimes$ can be used as abbreviation with the following meaning:

$$\bigotimes_{i=a}^{b} G_i = ((\cdots (G_a \otimes G_{a+1}) \otimes \cdots) \otimes G_b).$$

Draw the following graphs and determine their network characteristics (degree, node connectivity, edge connectivity, bisection bandwidth, and diameter):

(a) linear array of size 4 $\otimes$ linear array of size 2,
(b) two-dimensional mesh with 2×4 nodes $\otimes$ linear array of size 3,
(c) linear array of size 3 $\otimes$ complete graph with 4 nodes,
(d) $\bigotimes_{i=2}^{4}$ linear array of size i,
(e) $\bigotimes_{i=1}^{k}$ linear array of size 23. Draw the graph for $k = 4$, but determine the characteristics for general values of k.

Exercise 2.10 Consider a three-dimensional hypercube network and prove that E-cube routing is deadlock free for this network, see Sect. 2.6.1, p. 48.

Exercise 2.11 In the directory-based cache coherence protocol described in Sect. 2.7.3, p. 81, in case of a read miss with dirty bit 1, the processor which has

the requested cache block sends it to both the directory controller and the requesting processor. Instead, the owning processor could send the cache block to the directory controller and this one could forward the cache block to the requesting processor. Specify the details of this protocol.

Exercise 2.12 Consider the following sequence of memory accesses:

$$2, 3, 11, 16, 21, 13, 64, 48, 19, 11, 3, 22, 4, 27, 6, 11$$

Consider a cache of size 16 bytes. For the following configurations of the cache determine for each of the memory accesses in the sequence whether it leads to a cache hit or a cache miss. Show the resulting cache state that results after each access with the memory locations currently held in cache. Determine the resulting miss rate:

(a) direct-mapped cache with block size 1,
(b) direct-mapped cache with block size 4,
(c) two-way set-associative cache with block size 1, LRU replacement strategy,
(d) two-way set-associative cache with block size 4, LRU replacement strategy,
(e) fully associative cache with block size 1, LRU replacement,
(f) fully associative cache with block size 4, LRU replacement.

Exercise 2.13 Consider the MSI protocol from Fig. 2.35, p. 79, for a bus-based system with three processors P_1, P_2, P_3. Each processor has a direct-mapped cache. The following sequence of memory operations access two memory locations A and B which are mapped to the same cache line:

Processor	Action
P_1	write A, 4
P_3	write B, 8
P_2	read A
P_3	read A
P_3	write A, B
P_2	read A
P_1	read B
P_1	write B, 10

We assume that the variables are initialized to $A = 3$ and $B = 3$ and that the caches are initially empty. For each memory access determine

- the cache state of each processor after the memory operations,
- the content of the cache and the memory location for A and B,
- the processor actions (`PrWr`, `PrRd`) caused by the access, and
- the bus operations (`BusRd`, `BusRdEx`, `flush`) caused by the MSI protocol.

Exercise 2.14 Consider the following memory accesses of three processors P_1, P_2, P_3:

P_1	P_2	P_3
(1) $A = 1$; (2) $C = 1$;	(1) $B = A$;	(1) $D = C$;

The variables A, B, C, D are initialized to 0. Using the sequential consistency model, which values can the variables B and D have?

Exercise 2.15 Visit the Top500 web page at `www.top500.org` and determine important characteristics of the five fastest parallel computers, including number of processors or core, interconnection network, processors used, and memory hierarchy.

Exercise 2.16 Consider the following two realizations of a matrix traversal and computation:

```
for (j=0; j<1500; j++)
  for (i=0; i<1500; i++)
    x[i][j] = 2 · x[i][j];
```

```
for (i=0; i<1500; i++)
  for (j=0; j<1500; j++)
    x[i][j] = 2 · x[i][j];
```

We assume a cache of size 8 Kbytes with a large enough associativity so that no conflict misses occur. The cache line size is 32 bytes. Each entry of the matrix `x` occupies 8 bytes. The implementations of the loops are given in C which uses a row-major storage order for matrices. Compute the number of cache lines that must be loaded for each of the two loop nests. Which of the two loop nests leads to a better spatial locality?

Chapter 3
Parallel Programming Models

The coding of a parallel program for a given algorithm is strongly influenced by the parallel computing system to be used. The term *computing system* comprises all hardware and software components which are provided to the programmer and which form the programmer's view of the machine. The hardware architectural aspects have been presented in Chap. 2. The software aspects include the specific operating system, the programming language and the compiler, or the runtime libraries. The same parallel hardware can result in different views for the programmer, i.e., in different parallel computing systems when used with different software installations. A very efficient coding can usually be achieved when the specific hardware and software installation is taken into account. But in contrast to sequential programming there are many more details and diversities in parallel programming and a machine-dependent programming can result in a large variety of different programs for the same algorithm. In order to study more general principles in parallel programming, parallel computing systems are considered in a more abstract way with respect to some properties, like the organization of memory as shared or private. A systematic way to do this is to consider models which step back from details of single systems and provide an abstract view for the design and analysis of parallel programs.

3.1 Models for Parallel Systems

In the following, the types of models used for parallel processing according to [87] are presented. Models for parallel processing can differ in their level of abstraction. The four basic types are machine models, architectural models, computational models, and programming models. The **machine model** is at the lowest level of abstraction and consists of a description of hardware and operating system, e.g., the registers or the input and output buffers. Assembly languages are based on this level of models. **Architectural models** are at the next level of abstraction. Properties described at this level include the interconnection network of parallel platforms, memory organization, synchronous or asynchronous processing, and execution mode of single instructions by SIMD or MIMD.

T. Rauber, G. Rünger, *Parallel Programming*,
DOI 10.1007/978-3-642-04818-0_3, © Springer-Verlag Berlin Heidelberg 2010

The **computational model** (or model of computation) is at the next higher level of abstraction and offers an abstract or more formal model of a corresponding architectural model. It provides cost functions reflecting the time needed for the execution of an algorithm on the resources of a computer given by an architectural model. Thus, a computational model provides an analytical method for designing and evaluating algorithms. The complexity of an algorithm should reflect the performance on a real computer. For sequential computing, the RAM (random access machine) model is a computational model for the von Neumann architectural model. The RAM model describes a sequential computer by a memory and one processor accessing the memory. The memory consists of an unbounded number of memory locations each of which can contain an arbitrary value. The processor executes a sequential algorithm consisting of a sequence of instructions step by step. Each instruction comprises the load of data from memory into registers, the execution of an arithmetic or logical operation, and the storing of the result into memory. The RAM model is suitable for theoretical performance prediction although real computers have a much more diverse and complex architecture. A computational model for parallel processing is the PRAM (parallel random access machine) model, which is a generalization of the RAM model and is described in Chap. 4.

The **programming model** is at the next higher level of abstraction and describes a parallel computing system in terms of the semantics of the programming language or programming environment. A parallel programming model specifies the programmer's view on parallel computer by defining how the programmer can code an algorithm. This view is influenced by the architectural design and the language, compiler, or the runtime libraries and, thus, there exist many different parallel programming models even for the same architecture. There are several criteria by which the parallel programming models can differ:

- the level of parallelism which is exploited in the parallel execution (instruction level, statement level, procedural level, or parallel loops);
- the implicit or user-defined explicit specification of parallelism;
- the way how parallel program parts are specified;
- the execution mode of parallel units (SIMD or SPMD, synchronous or asynchronous);
- the modes and pattern of communication among computing units for the exchange of information (explicit communication or shared variables);
- synchronization mechanisms to organize computation and communication between parallel units.

Each parallel programming language or environment implements the criteria given above and there is a large number of different possibilities for combination. Parallel programming models provide methods to support the parallel programming.

The goal of a programming model is to provide a mechanism with which the programmer can specify parallel programs. To do so, a set of basic tasks must be supported. A parallel program specifies computations which can be executed in parallel. Depending on the programming model, the computations can be defined at

different levels: A computation can be (i) a sequence of *instructions* performing arithmetic or logical operations, (ii) a sequence of *statements* where each statement may capture several instructions, or (iii) a function or method invocation which typically consists of several statements. Many parallel programming models provide the concept of *parallel loops*; the iterations of a parallel loop are independent of each other and can therefore be executed in parallel, see Sect. 3.3.3 for an overview. Another concept is the definition of independent **tasks** (or *modules*) which can be executed in parallel and which are mapped to the processors of a parallel platform such that an efficient execution results. The mapping may be specified explicitly by the programmer or performed implicitly by a runtime library.

A parallel program is executed by the processors of a parallel execution environment such that on each processor one or multiple control flows are executed. Depending on the specific coordination, these control flows are referred to as **processes** or **threads**. The thread concept is a generalization of the process concept: A process can consist of several threads which share a common address space whereas each process works on a different address space. Which of these two concepts is more suitable for a given situation depends on the physical memory organization of the execution environment. The process concept is usually suitable for distributed memory organizations whereas the thread concept is typically used for shared memory machines, including multicore processors. In the following chapters, programming models based on the process or thread concept are discussed in more detail.

The processes or threads executing a parallel program may be created statically at program start. They may also be created during program execution according to the specific execution needs. Depending on the execution and synchronization modi supported by a specific programming model, there may or may not exist a hierarchical relation between the threads or processes. A fixed mapping from the threads or processes to the execution cores or processors of a parallel system may be used. In this case, a process or thread cannot be migrated to another processor or core during program execution. The partitioning into tasks and parallel execution modes for parallel programs are considered in more detail in Sects. 3.2–3.3.6. Data distributions for structured data types like vectors or matrices are considered in Sect. 3.4.

An important classification for parallel programming models is the *organization of the address space*. There are models with a shared or distributed address space, but there are also hybrid models which combine features of both memory organizations. The address space has a significant influence on the information exchange between the processes or threads. For a shared address space, shared variables are often used. Information exchange can be performed by write or read accesses of the processors or threads involved. For a distributed address space, each process has a local memory, but there is no shared memory via which information or data could be exchanged. Therefore, information exchange must be performed by additional message-passing operations to send or receive messages containing data or information. More details will be given in Sect. 3.5.

3.2 Parallelization of Programs

The parallelization of a given algorithm or program is typically performed on the basis of the programming model used. Independent of the specific programming model, typical steps can be identified to perform the parallelization. In this section, we will describe these steps. We assume that the computations to be parallelized are given in the form of a sequential program or algorithm. To transform the sequential computations into a parallel program, their control and data dependencies have to be taken into consideration to ensure that the parallel program produces the same results as the sequential program for all possible input values. The main goal is usually to reduce the program execution time as much as possible by using multiple processors or cores. The transformation into a parallel program is also referred to as **parallelization**. To perform this transformation in a systematic way, it can be partitioned into several steps:

1. **Decomposition of the computations**: The computations of the sequential algorithm are decomposed into tasks, and dependencies between the tasks are determined. The tasks are the smallest units of parallelism. Depending on the target system, they can be identified at different execution levels: instruction level, data parallelism, or functional parallelism, see Sect. 3.3. In principle, a task is a sequence of computations executed by a single processor or core. Depending on the memory model, a task may involve accesses to the shared address space or may execute message-passing operations. Depending on the specific application, the decomposition into tasks may be done in an initialization phase at program start (static decomposition), but tasks can also be created dynamically during program execution. In this case, the number of tasks available for execution can vary significantly during the execution of a program. At any point in program execution, the number of executable tasks is an upper bound on the available degree of parallelism and, thus, the number of cores that can be usefully employed. The goal of task decomposition is therefore to generate enough tasks to keep all cores busy at all times during program execution. But on the other hand, the tasks should contain enough computations such that the task execution time is large compared to the scheduling and mapping time required to bring the task to execution. The computation time of a task is also referred to as **granularity**: Tasks with many computations have a coarse-grained granularity, tasks with only a few computations are fine-grained. If task granularity is too fine-grained, the scheduling and mapping overhead is large and constitutes a significant amount of the total execution time. Thus, the decomposition step must find a good compromise between the number of tasks and their granularity.
2. **Assignment of tasks to processes or threads**: A process or a thread represents a flow of control executed by a physical processor or core. A process or thread can execute different tasks one after another. The number of processes or threads does not necessarily need to be the same as the number of physical processors or cores, but often the same number is used. The main goal of the assignment step is to assign the tasks such that a good **load balancing** results, i.e., each process

or thread should have about the same number of computations to perform. But the number of memory accesses (for shared address space) or communication operations for data exchange (for distributed address space) should also be taken into consideration. For example, when using a shared address space, it is useful to assign two tasks which work on the same data set to the same thread, since this leads to a good cache usage. The assignment of tasks to processes or threads is also called **scheduling**. For a static decomposition, the assignment can be done in the initialization phase at program start (static scheduling). But scheduling can also be done during program execution (dynamic scheduling).

3. **Mapping of processes or threads to physical processes or cores**: In the simplest case, each process or thread is mapped to a separate processor or core, also called execution unit in the following. If less cores than threads are available, multiple threads must be mapped to a single core. This mapping can be done by the operating system, but it could also be supported by program statements. The main goal of the mapping step is to get an equal utilization of the processors or cores while keeping communication between the processors as small as possible.

The parallelization steps are illustrated in Fig. 3.1.

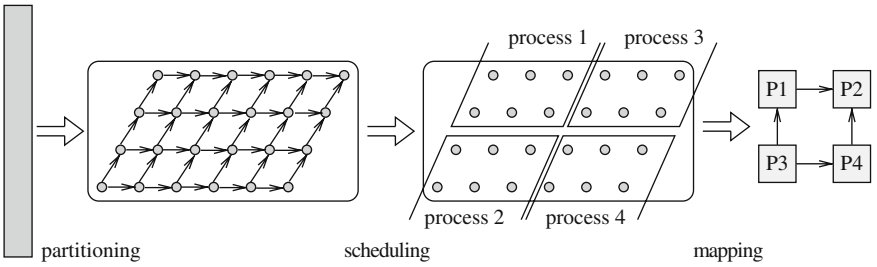


Fig. 3.1 Illustration of typical parallelization steps for a given sequential application algorithm. The algorithm is first split into tasks, and dependencies between the tasks are identified. These tasks are then assigned to processes by the scheduler. Finally, the processes are mapped to the physical processors P1, P2, P3, and P4

In general, a **scheduling algorithm** is a method to determine an efficient execution order for a set of tasks of a given duration on a given set of execution units. Typically, the number of tasks is much larger than the number of execution units. There may be dependencies between the tasks, leading to *precedence constraints*. Since the number of execution units is fixed, there are also *capacity constraints*. Both types of constraints restrict the schedules that can be used. Usually, the scheduling algorithm considers the situation that each task is executed sequentially by one processor or core (single-processor tasks). But in some models, a more general case is also considered which assumes that several execution units can be employed for a single task (parallel tasks), thus leading to a smaller task execution time. The overall goal of a scheduling algorithm is to find a schedule for the tasks which defines for each task a starting time and an execution unit such that the precedence and capacity constraints are fulfilled and such that a given objective function is optimized. Often,

the overall completion time (also called *makespan*) should be minimized. This is the time elapsed between the start of the first task and the completion of the last task of the program. For realistic situations, the problem of finding an optimal schedule is NP-complete or NP-hard [62]. A good overview of scheduling algorithms is given in [24].

Often, the number of processes or threads is adapted to the number of execution units such that each execution unit performs exactly one process or thread, and there is no migration of a process or thread from one execution unit to another during execution. In these cases, the terms "process" and "processor" or "thread" and "core" are used interchangeably.

3.3 Levels of Parallelism

The computations performed by a given program provide opportunities for parallel execution at different levels: instruction level, statement level, loop level, and function level. Depending on the level considered, tasks of different **granularity** result. Considering the instruction or statement level, fine-grained tasks result when a small number of instructions or statements are grouped to form a task. On the other hand, considering the function level, tasks are coarse-grained when the functions used to form a task comprise a significant amount of computations. On the loop level medium-grained tasks are typical, since one loop iteration usually consists of several statements. Tasks of different granularity require different scheduling methods to use the available potential of parallelism. In this section, we give a short overview of the available degree of parallelism at different levels and how it can be exploited in different programming models.

3.3.1 Parallelism at Instruction Level

Multiple instructions of a program can be executed in parallel at the same time, if they are independent of each other. In particular, the existence of one of the following **data dependencies** between instructions I_1 and I_2 inhibits their parallel execution:

- **Flow dependency** (also called *true dependency*): There is a flow dependency from instruction I_1 to I_2, if I_1 computes a result value in a register or variable which is then used by I_2 as operand.
- **Anti-dependency**: There is an anti-dependency from I_1 to I_2, if I_1 uses a register or variable as operand which is later used by I_2 to store the result of a computation.
- **Output dependency**: There is an output dependency from I_1 to I_2, if I_1 and I_2 use the same register or variable to store the result of a computation.

Figure 3.2 shows examples of the different dependency types [179]. In all three cases, instructions I_1 and I_2 cannot be executed in opposite order or in parallel,

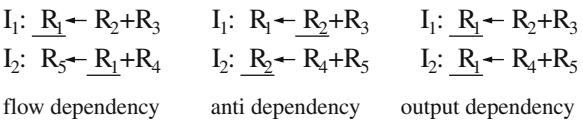


Fig. 3.2 Different types of data dependencies between instructions using registers $R_1, \ldots, R_5$. For each type, two instructions are shown which assign a new value to the registers on the *left-hand side* (represented by an *arrow*). The new value results by applying the operation on the *right-hand side* to the register operands. The register causing the dependence is *underlined*

since this would result in an erroneous computation: For the flow dependence, I_2 would use an old value as operand if the order is reversed. For the anti-dependence, I_1 would use the wrong value computed by I_2 as operand, if the order is reversed. For the output dependence, the subsequent instructions would use a wrong value for R_1, if the order is reversed. The dependencies between instructions can be illustrated by a data dependency graph. Figure 3.3 shows the data dependency graph for a sequence of instructions.

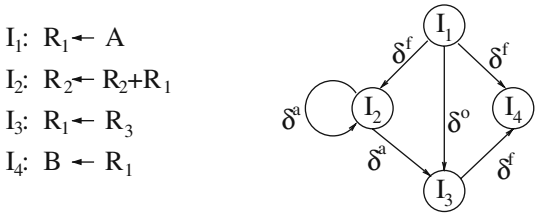


Fig. 3.3 Data dependency graph for a sequence I_1, I_2, I_3, I_4 of instructions using registers R_1, R_2, R_3 and memory addresses A, B. The edges representing a flow dependency are annotated with δ^f. Edges for anti-dependencies and output dependencies are annotated with δ^a and δ^o, respectively. There is a flow dependence from I_1 to I_2 and to I_4, since these two instructions use register R_1 as operand. There is an output dependency from I_1 to I_3, since both instructions use the same output register. Instruction I_2 has an anti-dependency to itself caused by R_2. The flow dependency from I_3 to I_4 is caused by R_1. Finally, there is an anti-dependency from I_2 to I_3 because of R_1

Superscalar processors with multiple functional units can execute several instructions in parallel. They employ a dynamic instruction scheduling realized in hardware, which extracts independent instructions from a sequential machine program by checking whether one of the dependence types discussed above exists. These independent instructions are then assigned to the functional units for execution. For VLIW processors, static scheduling by the compiler is used to identify independent instructions and to arrange a sequential flow of instructions in appropriate long instruction words such that the functional units are explicitly addressed. For both cases, a sequential program is used as input, i.e., no explicit specification of parallelism is used. Appropriate compiler techniques like software pipelining and trace scheduling can help to rearrange the instructions such that more parallelism can be extracted, see [48, 12, 7] for more details.

3.3.2 Data Parallelism

In many programs, the same operation must be applied to different elements of a larger data structure. In the simplest case, this could be an array structure. If the operations to be applied are independent of each other, this could be used for parallel execution: The elements of the data structure are distributed evenly among the processors and each processor performs the operation on its assigned elements. This form of parallelism is called **data parallelism** and can be used in many programs, especially from the area of scientific computing. To use data parallelism, sequential programming languages have been extended to **data-parallel programming languages**. Similar to sequential programming languages, *one single* control flow is used, but there are special constructs to express data-parallel operations on data structures like arrays. The resulting execution scheme is also referred to as SIMD model, see Sect. 2.2.

Often, data-parallel operations are only provided for arrays. A typical example is the *array assignments* of Fortran 90/95, see [49, 175, 122]. Other examples for data-parallel programming languages are C* and data-parallel C [82], PC++ [22], DINO [151], and High-Performance Fortran (HPF) [54, 57]. An example for an array assignment in Fortran 90 is

```
a(1:n) = b(0:n-1) + c(1:n).
```

The computations performed by this assignment are identical to those computed by the following loop:

```
for (i=1:n)
   a(i) = b(i-1) + c(i)
endfor.
```

Similar to other data-parallel languages, the semantics of an array assignment in Fortran 90 is defined as follows: First, all array accesses and operations on the right-hand side of the assignment are performed. After the complete right-hand side is computed, the actual assignment to the array elements on the left-hand side is performed. Thus, the following array assignment

```
a(1:n) = a(0:n-1) + a(2:n+1)
```

is not identical to the loop

```
for (i=1:n)
   a(i) = a(i-1) + a(i+1)
endfor.
```

The array assignment uses the old values of `a(0:n-1)` and `a(2:n+1)` whereas the `for` loop uses the old value only for `a(i+1)`; for `a(i-1)` the new value is used, which has been computed in the preceding iteration.

Data parallelism can also be exploited for MIMD models. Often, the SPMD model (**S**ingle **P**rogram **M**ultiple **D**ata) is used which means that *one* parallel program is executed by all processors in parallel. Program execution is performed *asynchronously* by the participating processors. Using the SPMD model, data parallelism results if each processor gets a part of a data structure for which it is responsible. For example, each processor could get a part of an array identified by a lower and an upper bound stored in private variables of the processor. The processor ID can be used to compute for each processor its part assigned. Different data distributions can be used for arrays, see Sect. 3.4 for more details. Figure 3.4 shows a part of an SPMD program to compute the scalar product of two vectors.

In practice, most parallel programs are SPMD programs, since they are usually easier to understand than general MIMD programs, but provide enough expressiveness to formulate typical parallel computation patterns. In principle, each processor can execute a different program part, depending on its processor ID. Most parallel programs shown in the rest of the book are SPMD programs.

Data parallelism can be exploited for both shared and distributed address spaces. For a distributed address space, the program data must be distributed among the processors such that each processor can access the data that it needs for its computations directly from its local memory. The processor is then called the *owner* of its local data. Often, the distribution of data and computation is done in the same way such that each processor performs the computations specified in the program on the

```
local_size = size/p;
local_lower = me * local_size;
local_upper = (me+1) * local_size - 1;
local_sum = 0.0;

for (i=local_lower; i<=local_upper; i++)
  local_sum += x[i] * y[i];

Reduce(&local_sum, &global_sum, 0, SUM);
```

Fig. 3.4 SPMD program to compute the scalar product of two vectors `x` and `y`. All variables are assumed to be private, i.e., each processor can store a different value in its local instance of a variable. The variable `p` is assumed to be the number of participating processors, `me` is the rank of the processor, starting from rank 0. The two arrays `x` and `y` with `size` elements each and the corresponding computations are distributed blockwise among the processors. The size of a data block of each processor is computed in `local_size`, the lower and upper bounds of the local data block are stored in `local_lower` and `local_upper`, respectively. For simplicity, we assume that `size` is a multiple of `p`. Each processor computes in `local_sum` the partial scalar product for its local data block of `x` and `y`. These partial scalar products are accumulated with the reduction function `Reduce()` at processor 0. Assuming a distribution address space, this reduction can be obtained by calling the MPI function `MPI_Reduce(&local_sum, &global_sum, 1, MPI_FLOAT, MPI_SUM, 0, MPI_COMM_WORLD)`, see Sect. 5.2

data that it stores in its local memory. This is called **owner-computes rule**, since the owner of the data performs the computations on this data.

3.3.3 Loop Parallelism

Many algorithms perform computations by iteratively traversing a large data structure. The iterative traversal is usually expressed by a loop provided by imperative programming languages. A loop is usually executed *sequentially* which means that the computations of the ith iteration are started not before all computations of the $(i-1)$th iteration are completed. This execution scheme is called *sequential loop* in the following. If there are no dependencies between the iterations of a loop, the iterations can be executed in arbitrary order, and they can also be executed in parallel by different processors. Such a loop is then called a *parallel loop*. Depending on their exact execution behavior, different types of parallel loops can be distinguished as will be described in the following [175, 12].

3.3.3.1 `forall` Loop

The body of a `forall` loop can contain one or several assignments to array elements. If a `forall` loop contains a single assignment, it is equivalent to an array assignment, see Sect. 3.3.2, i.e., the computations specified by the right-hand side of the assignment are first performed in any order, and then the results are assigned to their corresponding array elements, again in any order. Thus, the loop

```
forall (i = 1:n)
  a(i) = a(i-1) + a(i+1)
endforall
```

is equivalent to the array assignment

```
a(1:n) = a(0:n-1) + a(2:n+1)
```

in Fortran 90/95. If the `forall` loop contains multiple assignments, these are executed *one after another* as array assignments, such that the next array assignment is started not before the previous array assignment has been completed. A `forall` loop is provided in Fortran 95, but not in Fortran 90, see [122] for details.

3.3.3.2 `dopar` Loop

The body of a `dopar` loop may not only contain one or several assignments to array elements, but also other statements and even other loops. The iterations of a `dopar` loop are executed by multiple processors in parallel. Each processor executes its iterations in any order one after another. The instructions of each iteration are executed sequentially in program order, using the variable values of the initial state

before the dopar loop is started. Thus, variable updates performed in one iteration are not visible to the other iterations. After all iterations have been executed, the updates of the single iterations are combined and a new global state is computed. If two different iterations update the same variable, one of the two updates becomes visible in the new global state, resulting in a *non-deterministic* behavior.

The overall effect of forall and dopar loops with the same loop body may differ if the loop body contains more than one statement. This is illustrated by the following example [175].

Example We consider the following three loops:

```
for (i=1:4)              forall (i=1:4)          dopar (i=1:4)
  a(i)=a(i)+1              a(i)=a(i)+1             a(i)=a(i)+1
  b(i)=a(i-1)+a(i+1)       b(i)=a(i-1)+a(i+1)      b(i)=a(i-1)+a(i+1)
endfor                   endforall               enddopar
```

In the sequential for loop, the computation of b(i) uses the value of a(i-1) that has been computed in the preceding iteration and the value of a(i+1) valid before the loop. The two statements in the forall loop are treated as separate array assignments. Thus, the computation of b(i) uses for both a(i-1) and a(i+1) the new value computed by the first statement. In the dopar loop, updates in one iteration are not visible to the other iterations. Since the computation of b(i) does not use the value of a(i) that is computed in the same iteration, the old values are used for a(i-1) and a(i+1). The following table shows an example for the values computed:

Start values			After for loop	After forall loop	After dopar loop
a(0)	1				
a(1)	2	b(1)	4	5	4
a(2)	3	b(2)	7	8	6
a(3)	4	b(3)	9	10	8
a(4)	5	b(4)	11	11	10
a(5)	6				

□

A dopar loop in which an array element computed in an iteration is only used in that iteration is sometimes called doall loop. The iterations of such a doall loop are independent of each other and can be executed sequentially, or in parallel in any order without changing the overall result. Thus, a doall loop is a *parallel loop* whose iterations can be distributed arbitrarily among the processors and can be executed without synchronization. On the other hand, for a general dopar loop, it has to be made sure that the different iterations are separated, if a processor executes multiple iterations of the same loop. A processor is not allowed to use array values that it has computed in another iteration. This can be ensured by introducing temporary variables to store those array operands of the right-hand side that might cause

conflicts and using these temporary variables on the right-hand side. On the left-hand side, the original array variables are used. This is illustrated by the following example:

Example The following `dopar` loop

```
dopar (i=2:n-1)
  a(i) = a(i-1) + a(i+1)
enddopar
```

is equivalent to the following program fragment

```
doall (i=2:n-1)
  t1(i) = a(i-1)
  t2(i) = a(i+1)
enddoall
doall (i=2:n-1)
  a(i) = t1(i) + t2(i)
enddoall,
```

where `t1` and `t2` are temporary array variables. □

More information on parallel loops and their execution as well as on transformations to improve parallel execution can be found in [142, 175]. Parallel loops play an important role in programming environments like OpenMP, see Sect. 6.3 for more details.

3.3.4 Functional Parallelism

Many sequential programs contain program parts that are independent of each other and can be executed in parallel. The independent program parts can be single statements, basic blocks, loops, or function calls. Considering the independent program parts as tasks, this form of parallelism is called **task parallelism** or **functional parallelism**. To use task parallelism, the tasks and their dependencies can be represented as a **task graph** where the nodes are the tasks and the edges represent the dependencies between the tasks. A dependence graph is used for the conjugate gradient method discussed in Sect. 7.4. Depending on the programming model used, a single task can be executed sequentially by *one* processor, or in parallel by *multiple* processors. In the latter case, each task can be executed in a data-parallel way, leading to mixed task and data parallelism.

To determine an execution plan (schedule) for a given task graph on a set of processors, a starting time has to be assigned to each task such that the dependencies are fulfilled. Typically, a task cannot be started before all tasks which it depends on are finished. The goal of a scheduling algorithm is to find a schedule that minimizes the overall execution time, see also Sect. 4.3. Static and dynamic scheduling algorithms can be used. A *static* scheduling algorithm determines the assignment of tasks to processors deterministically at program start or at compile time. The assignment

may be based on an estimation of the execution time of the tasks, which might be obtained by runtime measurements or an analysis of the computational structure of the tasks, see Sect. 4.3. A detailed overview of static scheduling algorithms for different kinds of dependencies can be found in [24]. If the tasks of a task graph are *parallel tasks*, the scheduling problem is sometimes called *multiprocessor task scheduling*.

A *dynamic scheduling algorithm* determines the assignment of tasks to processors during program execution. Therefore, the schedule generated can be adapted to the observed execution times of the tasks. A popular technique for dynamic scheduling is the use of a **task pool** in which tasks that are ready for execution are stored and from which processors can retrieve tasks if they have finished the execution of their current task. After the completion of the task, all depending tasks in the task graph whose predecessors have been terminated can be stored in the task pool for execution. The task pool concept is particularly useful for shared address space machines since the task pool can be held in the global memory. The task pool concept is discussed further in Sect. 6.1 in the context of pattern programming. The implementation of task pools with Pthreads and their provision in Java is considered in more detail in Chap. 6. A detailed treatment of task pools is considered in [116, 159, 108, 93]. Information on the construction and scheduling of task graphs can be found in [18, 67, 142, 145]. The use of task pools for irregular applications is considered in [153]. Programming with multiprocessor tasks is supported by library-based approaches like Tlib [148].

Task parallelism can also be provided at language level for appropriate language constructs which specify the available degree of task parallelism. The management and mapping can then be organized by the compiler and the runtime system. This approach has the advantage that the programmer is only responsible for the specification of the degree of task parallelism. The actual mapping and adaptation to specific details of the execution platform is done by the compiler and runtime system, thus providing a clear separation of concerns. Some language approaches are based on *coordination languages* to specify the degree of task parallelism and dependencies between the tasks. Some approaches in this direction are TwoL (*Two Level parallelism*) [146], P3L (*Pisa Parallel Programming Language*) [138], and PCN (*Program Composition Notation*) [58]. A more detailed treatment can be found in [80, 46]. Many thread-parallel programs are based on the exploitation of functional parallelism, since each thread executes independent function calls. The implementation of thread parallelism will be considered in detail in Chap. 6.

3.3.5 Explicit and Implicit Representation of Parallelism

Parallel programming models can also be distinguished depending on whether the available parallelism, including the partitioning into tasks and specification of communication and synchronization, is represented explicitly in the program or not. The development of parallel programs is facilitated if no explicit representation must be included, but in this case an advanced compiler must be available to produce

efficient parallel programs. On the other hand, an explicit representation is more effort for program development, but the compiler can be much simpler. In the following, we briefly discuss both approaches. A more detailed treatment can be found in [160].

3.3.5.1 Implicit Parallelism

For the programmer, the simplest model results, when no explicit representation of parallelism is required. In this case, the program is mainly a specification of the computations to be performed, but no parallel execution order is given. In such a model, the programmer can concentrate on the details of the (sequential) algorithm to be implemented and does not need to care about the organization of the parallel execution. We give a short description of two approaches in this direction: parallelizing compilers and functional programming languages.

The idea of **parallelizing compilers** is to transform a sequential program into an efficient parallel program by using appropriate compiler techniques. This approach is also called *automatic parallelization*. To generate the parallel program, the compiler must first analyze the dependencies between the computations to be performed. Based on this analysis, the computation can then be assigned to processors for execution such that a good load balancing results. Moreover, for a distributed address space, the amount of communication should be reduced as much as possible, see [142, 175, 12, 6]. In practice, automatic parallelization is difficult to perform because dependence analysis is difficult for pointer-based computations or indirect addressing and because the execution time of function calls or loops with unknown bounds is difficult to predict at compile time. Therefore, automatic parallelization often produces parallel programs with unsatisfactory runtime behavior and, hence, this approach is not often used in practice.

Functional programming languages describe the computations of a program as the evaluation of mathematical functions without side effects; this means the evaluation of a function has the only effect that the output value of the function is computed. Thus, calling a function twice with the same input argument values always produces the same output value. Higher-order functions can be used; these are functions which use other functions as arguments and yield functions as arguments. Iterative computations are usually expressed by recursion. The most popular functional programming language is Haskell, see [94, 170, 20]. Function evaluation in functional programming languages provides potential for parallel execution, since the arguments of the function can always be evaluated in parallel. This is possible because of the lack of side effects. The problem of an efficient execution is to extract the parallelism at the right level of recursion: On the upper level of recursion, a parallel evaluation of the arguments may not provide enough potential for parallelism. On a lower level of recursion, the available parallelism may be too fine-grained, thus making an efficient assignment to processors difficult. In the context of multicore processors, the degree of parallelism provided at the upper level of recursion may be enough to efficiently supply a few cores with computations. The advantage of using

functional languages would be that new language constructs are not necessary to enable a parallel execution as is the case for non-functional programming languages.

3.3.5.2 Explicit Parallelism with Implicit Distribution

Another class of parallel programming models comprises models which require an explicit representation of parallelism in the program, but which do not demand an explicit distribution and assignment to processes or threads. Correspondingly, no explicit communication or synchronization is required. For the compiler, this approach has the advantage that the available degree of parallelism is specified in the program and does not need to be retrieved by a complicated data dependence analysis. This class of programming models includes parallel programming languages which extend sequential programming languages by *parallel loops* with independent iterations, see Sect. 3.3.3.

The parallel loops specify the available parallelism, but the exact assignments of loop iterations to processors is not fixed. This approach has been taken by the library OpenMP where parallel loops can be specified by compiler directives, see Sect. 6.3 for more details on OpenMP. High-Performance Fortran (HPF) [54] has been another approach in this direction which adds constructs for the specification of array distributions to support the compiler in the selection of an efficient data distribution, see [103] on the history of HPF.

3.3.5.3 Explicit Distribution

A third class of parallel programming models requires not only an explicit representation of parallelism, but also an explicit partitioning into tasks or an explicit assignment of work units to threads. The mapping to processors or cores as well as communication between processors is implicit and does not need to be specified. An example for this class is the BSP (bulk synchronous parallel) programming model which is based on the BSP computation model described in more detail in Sect. 4.5.2 [88, 89]. An implementation of the BSP model is BSPLib. A BSP program is explicitly partitioned into threads, but the assignment of threads to processors is done by the BSPLib library.

3.3.5.4 Explicit Assignment to Processors

The next class captures parallel programming models which require an explicit partitioning into tasks or threads and also need an explicit assignment to processors. But the communication between the processors does not need to be specified. An example for this class is the coordination language Linda [27, 26] which replaces the usual point-to-point communication between processors by a **tuple space** concept. A tuple space provides a global pool of data in which data can be stored and from which data can be retrieved. The following three operations are provided to access the tuple space:

- **in**: read and remove a tuple from the tuple space;
- **read**: read a tuple from the tuple space without removing it;
- **out**: write a tuple in the tuple space.

A tuple to be retrieved from the tuple space is identified by specifying required values for a part of the data fields which are interpreted as a key. For distributed address spaces, the access operations to the tuple space must be implemented by communication operations between the processes involved: If in a Linda program, a process A writes a tuple into the tuple space which is later retrieved by a process B, a communication operation from process A (`send`) to process B (`recv`) must be generated. Depending on the execution platform, this communication may produce a significant amount of overhead. Other approaches based on a tuple space are TSpaces from IBM and JavaSpaces [21] which is part of the Java Jini technology.

3.3.5.5 Explicit Communication and Synchronization

The last class comprises programming models in which the programmer must specify all details of a parallel execution, including the required communication and synchronization operations. This has the advantage that a standard compiler can be used and that the programmer can control the parallel execution explicitly with all the details. This usually provides efficient parallel programs, but it also requires a significant amount of work for program development. Programming models belonging to this class are message-passing models like MPI, see Chap. 5, as well as thread-based models like Pthreads, see Chap. 6.

3.3.6 Parallel Programming Patterns

Parallel programs consist of a collection of tasks that are executed by processes or threads on multiple processors. To structure a parallel program, several forms of organizations can be used which can be captured by specific programming patterns. These patterns provide specific coordination structures for processes or threads, which have turned out to be effective for a large range of applications. We give a short overview of useful programming patterns in the following. More information and details on the implementation in specific environments can be found in [120]. Some of the patterns are presented as programs in Chap. 6.

3.3.6.1 Creation of Processes or Threads

The creation of processes or threads can be carried out statically or dynamically. In the static case, a fixed number of processes or threads is created at program start. These processes or threads exist during the entire execution of the parallel program and are terminated when program execution is finished. An alternative approach is to allow creation and termination of processes or threads dynamically at arbitrary points during program execution. At program start, a single process or thread is

active and executes the main program. In the following, we describe well-known parallel programming patterns. For simplicity, we restrict our attention to the use of threads, but the patterns can as well be applied to the coordination of processes.

3.3.6.2 Fork–Join

The fork–join construct is a simple concept for the creation of processes or threads [30] which was originally developed for process creation, but the pattern can also be used for threads. Using the concept, an existing thread T creates a number of child threads $T_1, \ldots, T_m$ with a `fork` statement. The child threads work in parallel and execute a given program part or function. The creating parent thread T can execute the same or a different program part or function and can then wait for the termination of $T_1, \ldots, T_m$ by using a `join` call.

The fork–join concept can be provided as a language construct or as a library function. It is usually provided for shared address space, but can also be used for distributed address space. The fork–join concept is, for example, used in OpenMP for the creation of threads executing a parallel loop, see Sect. 6.3 for more details. The **spawn** and **exit** operations provided by message-passing systems like MPI-2, see Sect. 5, provide a similar action pattern as fork–join. The concept of fork–join is simple, yet flexible, since by a nested use, arbitrary structures of parallel activities can be built. Specific programming languages and environments provide specific variants of the pattern, see Chap. 6 for details on Pthreads and Java threads.

3.3.6.3 Parbegin–Parend

A similar pattern as fork–join for thread creation and termination is provided by the **parbegin–parend** construct which is sometimes also called **cobegin–coend**. The construct allows the specification of a sequence of statements, including function calls, to be executed by a set of processors in parallel. When an executing thread reaches a parbegin–parend construct, a set of threads is created and the statements of the construct are assigned to these threads for execution. The statements following the parbegin–parend construct are executed not before all these threads have finished their work and have been terminated. The parbegin–parend construct can be provided as a language construct or by compiler directives. An example is the construct of parallel sections in OpenMP, see Sect. 6.3 for more details.

3.3.6.4 SPMD and SIMD

The SIMD (**s**ingle-**i**nstruction, **m**ultiple-**d**ata) and SPMD (**s**ingle-**p**rogram, **m**ultiple-**d**ata) programming models use a (fixed) number of threads which apply the same program to different data. In the SIMD approach, the single instructions are executed synchronously by the different threads on different data. This is sometimes called *data parallelism in the strong sense*. SIMD is useful if the same instruction must be applied to a large set of data, as is often the case for graphics applications. Therefore,

graphics processors often provide SIMD instructions, and some standard processors also provide SIMD extensions.

In the SPMD approach, the different threads work asynchronously with each other and different threads may execute different parts of the parallel program. This effect can be caused by different speeds of the executing processors or by delays of the computations because of slower access to global data. But the program could also contain control statements to assign different program parts to different threads. There is no implicit synchronization of the executing threads, but synchronization can be achieved by explicit synchronization operations. The SPMD approach is one of the most popular models for parallel programming. MPI is based on this approach, see Sect. 5, but thread-parallel programs are usually also SPMD programs.

3.3.6.5 Master–Slave or Master–Worker

In the SIMD and SPMD models, all threads have equal rights. In the master–slave model, also called master–worker model, there is one master which controls the execution of the program. The master thread often executes the main function of a parallel program and creates worker threads at appropriate program points to perform the actual computations, see Fig. 3.5 (left) for an illustration. Depending on the specific system, the worker threads may be created statically or dynamically. The assignment of work to the worker threads is usually done by the master thread, but worker threads could also generate new work for computation. In this case, the master thread would only be responsible for coordination and could, e.g., perform initializations, timings, and output operations.

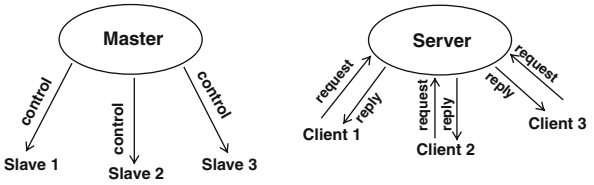


Fig. 3.5 Illustration of the master–slave model (*left*) and the client–server model (*right*)

3.3.6.6 Client–Server

The coordination of parallel programs according to the client–server model is similar to the general MPMD (**m**ultiple-**p**rogram **m**ultiple-**d**ata) model. The client–server model originally comes from distributed computing where multiple client computers have been connected to a mainframe which acts as a server and provides responses to access requests to a database. On the server side, parallelism can be used by computing requests from different clients concurrently or even by using multiple threads to compute a single request if this includes enough work.

When employing the client–server model for the structuring of parallel programs, multiple client threads are used which generate requests to a server and then perform some computations on the result, see Fig. 3.5 (right) for an illustration. After having processed a request of a client, the server delivers the result back to the client. The client–server model can be applied in many variations: There may be several server threads or the threads of a parallel program may play the role of both clients and servers, generating requests to other threads and processing requests from other threads. Section 6.1.8 shows an example for a Pthreads program using the client–server model. The client–server model is important for parallel programming in heterogeneous systems and is also often used in grid computing and cloud computing.

3.3.6.7 Pipelining

The pipelining model describes a special form of coordination of different threads in which data elements are forwarded from thread to thread to perform different processing steps. The threads are logically arranged in a predefined order, $T_1, \ldots, T_p$, such that thread T_i receives the output of thread T_{i-1} as input and produces an output which is submitted to the next thread T_{i+1} as input, $i = 2, \ldots, p-1$. Thread T_1 receives its input from another program part and thread T_p provides its output to another program part. Thus, each of the pipeline threads processes a stream of input data in sequential order and produces a stream of output data. Despite the dependencies of the processing steps, the pipeline threads can work in parallel by applying their processing step to different data.

The pipelining model can be considered as a special form of functional decomposition where the pipeline threads process the computations of an application algorithm one after another. A parallel execution is obtained by partitioning the data into a stream of data elements which flow through the pipeline stages one after another. At each point in time, different processing steps are applied to different elements of the data stream. The pipelining model can be applied for both shared and distributed address spaces. In Sect. 6.1, the pipelining pattern is implemented as Pthreads program.

3.3.6.8 Task Pools

In general, a task pool is a data structure in which tasks to be performed are stored and from which they can be retrieved for execution. A task comprises computations to be executed and a specification of the data to which the computations should be applied. The computations are often specified as a function call. A *fixed* number of threads is used for the processing of the tasks. The threads are created at program start by the main thread and they are terminated not before all tasks have been processed. For the threads, the task pool is a common data structure which they can access to retrieve tasks for execution, see Fig. 3.6 (left) for an illustration. During the processing of a task, a thread can generate new tasks and insert them into the

This figure will be printed in b/w

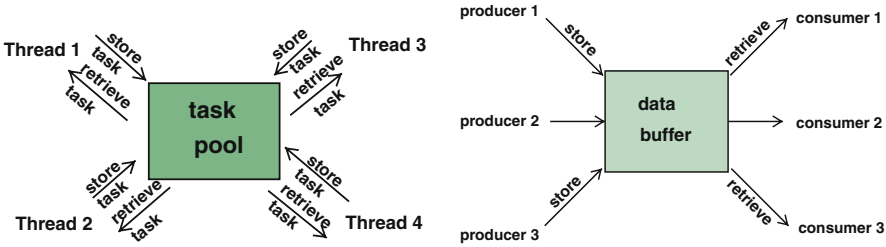


Fig. 3.6 Illustration of a task pool (*left*) and a producer–consumer model (*right*)

task pool. Access to the task pool must be synchronized to avoid race conditions. Using a task-based execution, the execution of a parallel program is finished, when the task pool is empty and when each thread has terminated the processing of its last task. Task pools provide a flexible execution scheme which is especially useful for adaptive and irregular applications for which the computations to be performed are not fixed at program start. Since a fixed number of threads is used, the overhead for thread creation is independent of the problem size and the number of tasks to be processed.

Flexibility is ensured, since tasks can be generated dynamically at any point during program execution. The actual task pool data structure could be provided by the programming environment used or could be included in the parallel program. An example for the first case is the `Executor` interface of Java, see Sect. 6.2 for more details. A simple task pool implementation based on a shared data structure is described in Sect. 6.1.6 using Pthreads. For fine-grained tasks, the overhead of retrieval and insertion of tasks from or into the task pool becomes important, and sophisticated data structures should be used for the implementation, see [93] for more details.

3.3.6.9 Producer–Consumer

The producer–consumer model distinguishes between producer threads and consumer threads. Producer threads produce data which are used as input by consumer threads. For the transfer of data from producer threads to consumer threads, a common data structure is used, which is typically a data buffer of fixed length and which can be accessed by both types of threads. Producer threads store the data elements generated into the buffer, consumer threads retrieve data elements from the buffer for further processing, see Fig. 3.6 (right) for an illustration. A producer thread can only store data elements into the buffer, if this is not full. A consumer thread can only retrieve data elements from the buffer, if this is not empty. Therefore, synchronization has to be used to ensure a correct coordination between producer and consumer threads. The producer–consumer model is considered in more detail in Sect. 6.1.9 for Pthreads and Sect. 6.2.3 for Java threads.

3.4 Data Distributions for Arrays

Many algorithms, especially from numerical analysis and scientific computing, are based on vectors and matrices. The corresponding programs use one-, two-, or higher dimensional arrays as basic data structures. For those programs, a straightforward parallelization strategy decomposes the array-based data into subarrays and assigns the subarrays to different processors. The decomposition of data and the mapping to different processors is called **data distribution**, **data decomposition**, or **data partitioning**. In a parallel program, the processors perform computations only on their part of the data.

Data distributions can be used for parallel programs for distributed as well as for shared memory machines. For distributed memory machines, the data assigned to a processor reside in its local memory and can only be accessed by this processor. Communication has to be used to provide data to other processors. For shared memory machines, all data reside in the same shared memory. Still a data decomposition is useful for designing a parallel program since processors access different parts of the data and conflicts such as race conditions or critical regions are avoided. This simplifies the parallel programming and supports a good performance. In this section, we present regular data distributions for arrays, which can be described by a mapping from array indices to processor numbers. The set of processors is denoted as $P = \{P_1, \ldots, P_p\}$.

3.4.1 Data Distribution for One-Dimensional Arrays

For one-dimensional arrays the blockwise and the cyclic distribution of array elements are typical data distributions. For the formulation of the mapping, we assume that the enumeration of array elements starts with 1; for an enumeration starting with 0 the mappings have to be modified correspondingly.

The **blockwise data distribution** of an array $v = (v_1, \ldots, v_n)$ of length n cuts the array into p blocks with $\lceil n/p \rceil$ consecutive elements each. Block j, $1 \leq j \leq p$, contains the consecutive elements with indices $(j-1) \cdot \lceil n/p \rceil + 1, \ldots, j \cdot \lceil n/p \rceil$ and is assigned to processor P_j. When n is not a multiple of p, the last block contains less than $\lceil n/p \rceil$ elements. For $n = 14$ and $p = 4$ the following blockwise distribution results:

P_1: owns v_1, v_2, v_3, v_4,
P_2: owns v_5, v_6, v_7, v_8,
P_3: owns $v_9, v_{10}, v_{11}, v_{12}$,
P_4: owns v_{13}, v_{14}.

Alternatively, the first $n \bmod p$ processors get $\lceil n/p \rceil$ elements and all other processors get $\lfloor n/p \rfloor$ elements.

The **cyclic data distribution** of a one-dimensional array assigns the array elements in a round robin way to the processors so that array element v_i is assigned to processor $P_{(i-1) \bmod p + 1}$, $i = 1, \ldots, n$. Thus, processor P_j owns the array elements

$j, j+p, \ldots, j+p\cdot(\lceil n/p\rceil-1)$ for $j \leq n \bmod p$ and $j, j+p, \ldots, j+p\cdot(\lceil n/p\rceil-2)$ for $n \bmod p < j \leq p$. For the example $n = 14$ and $p = 4$ the cyclic data distribution

P_1: owns v_1, v_5, v_9, v_{13},
P_2: owns v_2, v_6, v_{10}, v_{14},
P_3: owns v_3, v_7, v_{11},
P_4: owns v_4, v_8, v_{12}

results, where P_j for $1 \leq j \leq 2 = 14 \bmod 4$ owns the elements $j, j+4, j+4*2, j+4*(4-1)$ and P_j for $2 < j \leq 4$ owns the elements $j, j+4, j+4*(4-2)$.

The **block–cyclic data distribution** is a combination of the blockwise and cyclic distributions. Consecutive array elements are structured into blocks of size b, where $b \ll n/p$ in most cases. When n is not a multiple of b, the last block contains less than b elements. The blocks of array elements are assigned to processors in a round robin way. Figure 3.7a shows an illustration of the array decompositions for one-dimensional arrays.

3.4.2 Data Distribution for Two-Dimensional Arrays

For two-dimensional arrays, combinations of blockwise and cyclic distributions in only one or both dimensions are used.

For the distribution in one dimension, columns or rows are distributed in a blockwise, cyclic, or block–cyclic way. The blockwise columnwise (or rowwise) distribution builds p blocks of contiguous columns (or rows) of equal size and assigns block i to processor P_i, $i = 1, \ldots, p$. When n is not a multiple of p, the same adjustment as for one-dimensional arrays is used. The cyclic columnwise (or rowwise) distribution assigns columns (or rows) in a round robin way to processors and uses the adjustments of the last blocks as described for the one-dimensional case, when n is not a multiple of p. The block–cyclic columnwise (or rowwise) distribution forms blocks of contiguous columns (or rows) of size b and assigns these blocks in a round robin way to processors. Figure 3.7b illustrates the distribution in one dimension for two-dimensional arrays.

A distribution of array elements of a two-dimensional array of size $n_1 \times n_2$ in both dimensions uses **checkerboard distributions** which distinguish between blockwise cyclic and block–cyclic checkerboard patterns. The processors are arranged in a virtual mesh of size $p_1 \cdot p_2 = p$ where p_1 is the number of rows and p_2 is the number of columns in the mesh. Array elements (k, l) are mapped to processors $P_{i,j}$, $i = 1, \ldots, p_1$, $j = 1, \ldots, p_2$.

In the **blockwise checkerboard distribution**, the array is decomposed into $p_1 \cdot p_2$ blocks of elements where the row dimension (first index) is divided into p_1 blocks and the column dimension (second index) is divided into p_2 blocks. Block (i, j), $1 \leq i \leq p_1$, $1 \leq j \leq p_2$, is assigned to the processor with position (i, j) in the processor mesh. The block sizes depend on the number of rows and columns of the array. Block (i, j) contains the array elements (k, l) with $k = (i-1)\cdot\lceil n_1/p_1\rceil+1, \ldots, i\cdot\lceil n_1/p_1\rceil$ and $l = (j-1)\cdot\lceil n_2/p_2\rceil+1, \ldots, j\cdot\lceil n_2/p_2\rceil$. Figure 3.7c shows an example for $n_1 = 4$, $n_2 = 8$, and $p_1 \cdot p_2 = 2 \cdot 2 = 4$.

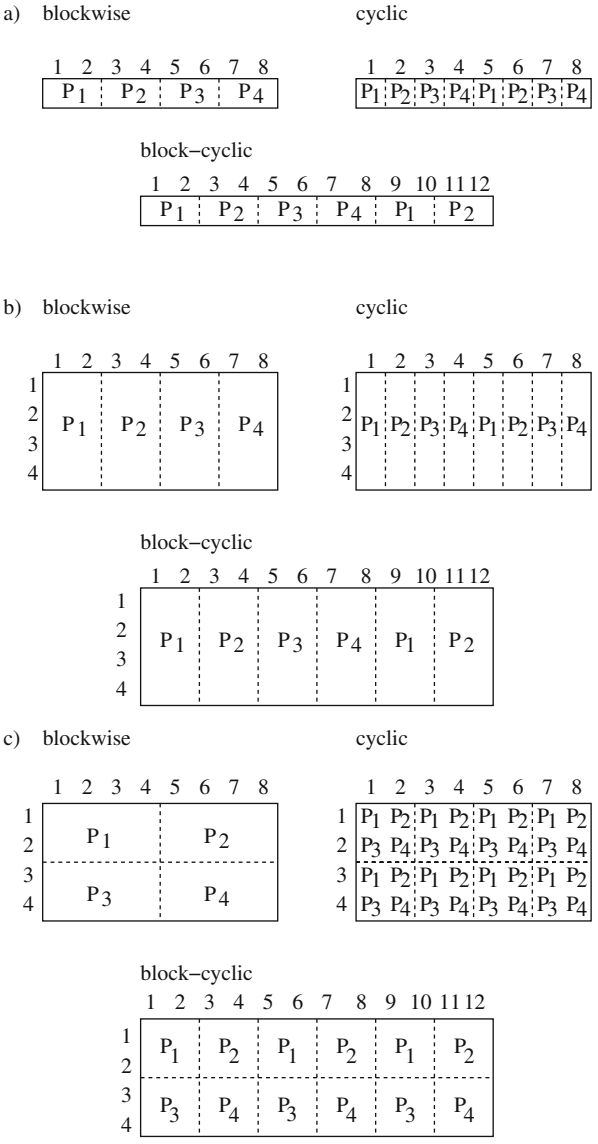


Fig. 3.7 Illustration of the data distributions for arrays: (**a**) for one-dimensional arrays, (**b**) for two-dimensional arrays within one of the dimensions, and (**c**) for two-dimensional arrays with checkerboard distribution

The **cyclic checkerboard distribution** assigns the array elements in a round robin way in both dimensions to the processors in the processor mesh so that a cyclic assignment of row indices $k = 1, \ldots, n_1$ to mesh rows $i = 1, \ldots, p_1$ and a cyclic assignment of column indices $l = 1, \ldots, n_2$ to mesh columns $j = 1, \ldots, p_2$ result. Array element (k, l) is thus assigned to the processor with mesh position

$((k-1) \bmod p_1 + 1, (l-1) \bmod p_2 + 1)$. When n_1 and n_2 are multiples of p_1 and p_2, respectively, the processor at position (i, j) owns all array elements (k, l) with $k = i + s \cdot p_1$ and $l = j + t \cdot p_2$ for $0 \leq s < n_1/p_1$ and $0 \leq t < n_2/p_2$. An alternative way to describe the cyclic checkerboard distribution is to build blocks of size $p_1 \times p_2$ and to map element (i, j) of each block to the processor at position (i, j) in the mesh. Figure 3.7c shows a cyclic checkerboard distribution with $n_1 = 4, n_2 = 8, p_1 = 2$, and $p_2 = 2$. When n_1 or n_2 is not a multiple of p_1 or p_2, respectively, the cyclic distribution is handled as in the one-dimensional case.

The **block–cyclic checkerboard distribution** assigns blocks of size $b_1 \times b_2$ cyclically in both dimensions to the processors in the following way: Array element (m, n) belongs to the block (k, l), with $k = \lceil m/b_1 \rceil$ and $l = \lceil n/b_2 \rceil$. Block (k, l) is assigned to the processor at mesh position $((k-1) \bmod p_1 + 1, (l-1) \bmod p_2 + 1)$. The cyclic checkerboard distribution can be considered as a special case of the block–cyclic distribution with $b_1 = b_2 = 1$, and the blockwise checkerboard distribution can be considered as a special case with $b_1 = n_1/p_1$ and $b_2 = n_2/p_2$. Figure 3.7c illustrates the block–cyclic distribution for $n_1 = 4, n_2 = 12, p_1 = 2$, and $p_2 = 2$.

3.4.3 Parameterized Data Distribution

A data distribution is defined for a d-dimensional array A with index set $I_A \subset \mathbb{N}^d$. The size of the array is $n_1 \times \cdots \times n_d$ and the array elements are denoted as $A[i_1, \ldots, i_d]$ with an index $\mathbf{i} = (i_1, \ldots, i_d) \in I_A$. Array elements are assigned to p processors which are arranged in a d-dimensional mesh of size $p_1 \times \cdots \times p_d$ with $p = \prod_{i=1}^{d} p_i$. The data distribution of A is given by a **distribution function** $\gamma_A : I_A \subset \mathbb{N}^d \rightarrow 2^P$, where 2^P denotes the power set of the set of processors P. The meaning of γ_A is that the array element $A[i_1, \ldots, i_d]$ with $\mathbf{i} = (i_1, \ldots, i_d)$ is assigned to all processors in $\gamma_A(\mathbf{i}) \subseteq P$, i.e., array element $A[\mathbf{i}]$ can be assigned to more than one processor. A data distribution is called **replicated**, if $\gamma_A(\mathbf{i}) = P$ for all $\mathbf{i} \in I_A$. When each array element is uniquely assigned to a processor, then $|\gamma_A(\mathbf{i})| = 1$ for all $\mathbf{i} \in I_A$; examples are the block–cyclic data distribution described above. The function $L(\gamma_A) : P \rightarrow 2^{I_A}$ delivers all elements assigned to a specific processor, i.e.,

$$\mathbf{i} \in L(\gamma_A)(q) \quad \text{if and only if} \quad q \in \gamma_A(\mathbf{i}).$$

Generalizations of the block–cyclic distributions in the one- or two-dimensional case can be described by a distribution vector in the following way. The array elements are structured into blocks of size $b_1, \ldots, b_d$ where b_i is the block size in dimension i, $i = 1, \ldots, d$. The array element $A[i_1, \ldots, i_d]$ is contained in block $(k_1, \ldots, k_d)$ with $k_j = \lceil i_j/b_j \rceil$ for $1 \leq j \leq d$. The block $(k_1, \ldots, k_d)$ is then assigned to the processor at mesh position $((k_1 - 1) \bmod p_1 + 1, \ldots, (k_d - 1) \bmod p_d + 1)$. This block–cyclic distribution is called **parameterized data distribution** with distribution vector

$$((p_1, b_1), \ldots, (p_d, b_d)) . \tag{3.1}$$

This vector uniquely determines a block–cyclic data distribution for a d-dimensional array of arbitrary size. The blockwise and the cyclic distributions of a d-dimensional array are special cases of this distribution. Parameterized data distributions are used in the applications of later sections, e.g., the Gaussian elimination in Sect. 7.1.

3.5 Information Exchange

To control the coordination of the different parts of a parallel program, information must be exchanged between the executing processors. The implementation of such an information exchange strongly depends on the memory organization of the parallel platform used. In the following, we give a first overview on techniques for information exchange for shared address space in Sect. 3.5.1 and for distributed address space in Sect. 3.5.2. More details will be discussed in the following chapters. As example, parallel matrix–vector multiplication is considered for both memory organizations in Sect. 3.6.

3.5.1 Shared Variables

Programming models with a shared address space are based on the existence of a global memory which can be accessed by all processors. Depending on the model, the executing control flows may be referred to as *processes* or *threads*, see Sect. 3.7 for more details. In the following, we will use the notation *threads*, since this is more common for shared address space models. Each thread will be executed by one processor or by one core for multicore processors. Each thread can access shared data in the global memory. Such shared data can be stored in **shared variables** which can be accessed as normal variables. A thread may also have private data stored in **private variables**, which cannot be accessed by other threads. There are different ways how parallel program environments define shared or private variables. The distinction between shared and private variables can be made by using annotations like `shared` or `private` when declaring the variables. Depending on the programming model, there can also be declaration rules which can, for example, define that global variables are always shared and local variables of functions are always private. To allow a coordinated access to a shared variable by multiple threads, synchronization operations are provided to ensure that concurrent accesses to the same variable are synchronized. Usually, a **sequentialization** is performed such that concurrent accesses are done one after another. Chapter 6 considers programming models and techniques for shared address spaces in more detail and describes different systems, like Pthreads, Java threads, and OpenMP. In the current section, a few basic concepts are given for a first overview.

A central concept for information exchange in shared address space is the use of shared variables. When a thread T_1 wants to transfer data to another thread T_2, it stores the data in a shared variable such that T_2 obtains the data by reading this shared variable. To ensure that T_2 reads the variable not before T_1 has written the appropriate data, a **synchronization operation** is used. T_1 stores the data into the shared variable before the corresponding synchronization point and T_2 reads the data after the synchronization point.

When using shared variables, multiple threads accessing the same shared variable by a read or write at the same time must be avoided, since this may lead to race conditions. The term **race condition** describes the effect that the result of a parallel execution of a program part by multiple execution units depends on the order in which the statements of the program part are executed by the different units. In the presence of a race condition it may happen that the computation of a program part leads to different results, depending on whether thread T_1 executes the program part before T_2 or vice versa. Usually, race conditions are undesirable, since the relative execution speed of the threads may depend on many factors (like execution speed of the executing cores or processors, the occurrence of interrupts, or specific values of the input data) which cannot be influenced by the programmer. This may lead to **non-deterministic behavior**, since, depending on the execution order, different results are possible, and the exact outcome cannot be predicted.

Program parts in which concurrent accesses to shared variables by multiple threads may occur, thus holding the danger of the occurrence of inconsistent values, are called **critical sections**. An error-free execution can be ensured by letting only one thread at a time execute a critical section. This is called **mutual exclusion**. Programming models for shared address space provide mechanisms to ensure mutual exclusion. The techniques used have originally been developed for multi-tasking operating systems and have later been adapted to the needs of parallel programming environments. For a concurrent access of shared variables, race conditions can be avoided by a **lock mechanism**, which will be discussed in more detail in Sect. 3.7.3.

3.5.2 Communication Operations

In programming models with a distributed address space, exchange of data and information between the processors is performed by *communication operations* which are *explicitly* called by the participating processors. The execution of such a communication operation causes one processor to receive data that is stored in the local memory of another processor. The actual data exchange is realized by the transfer of messages between the participating processors. The corresponding programming models are therefore called **message-passing programming models**.

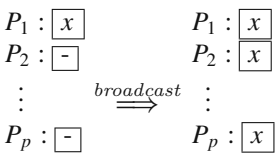
To send a message from one processor to another, one send and one receive operations have to be used as a pair. A send operation sends a data block from the local address space of the executing processor to another processor as specified by the operation. A receive operation receives a data block from another processor and

stores it in the local address space of the executing processor. This kind of data exchange is also called *point-to-point communication*, since there is exactly one send point and one receive point. Additionally, **global communication operations** are often provided in which a larger set of processors is involved. These global communication operations typically capture a set of regular communication patterns often used in parallel programs [19, 100].

3.5.2.1 A Set of Communication Operations

In the following, we consider a typical set of global communication operations which will be used in the following chapters to describe parallel implementations for platforms with a distributed address space [19]. We consider p identical processors $P_1, \ldots, P_p$ and use the index i, $i \in \{1, \ldots, p\}$, as processor rank to identify the processor P_i.

- **Single transfer**: For a single transfer operation, a processor P_i (sender) sends a message to processor P_j (receiver) with $j \neq i$. Only these two processors participate in this operation. To perform a single transfer operation, P_i executes a send operation specifying a send buffer in which the message is provided as well as the processor rank of the receiving processor. The receiving processor P_j executes a corresponding receive operation which specifies a receive buffer to store the received message as well as the processor rank of the processor from which the message should be received. For each send operation, there must be a corresponding receive operation, and vice versa. Otherwise, deadlocks may occur, see Sects. 3.7.4.2 and 5.1.1 for more details. Single transfer operations are the basis of each communication library. In principle, any communication pattern can be assembled with single transfer operations. For regular communication patterns, it is often beneficial to use global communication operations, since they are typically easier to use and more efficient.
- **Single-broadcast**: For a single-broadcast operation, a specific processor P_i sends the *same* data block to all other processors. P_i is also called *root* in this context. The effect of a single-broadcast operation with processor P_1 as root and message x can be illustrated as follows:



Before the execution of the broadcast, the message x is only stored in the local address space of P_1. After the execution of the operation, x is also stored in the local address space of all other processors. To perform the operation, each processor explicitly calls a broadcast operation which specifies the root processor of the broadcast. Additionally, the root processor specifies a send buffer in which

the broadcast message is provided. All other processors specify a receive buffer in which the message should be stored upon receipt.

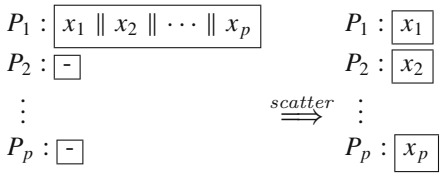
- **Single-accumulation**: For a single-accumulation operation, each processor provides a block of data with the same type and size. By performing the operation, a given reduction operation is applied element by element to the data blocks provided by the processors, and the resulting accumulated data block of the same length is collected at a specific root processor P_i. The reduction operation is a binary operation which is associative and commutative. The effect of a single-accumulation operation with root processor P_1 to which each processor P_i provides a data block x_i for $i = 1, \ldots, p$ can be illustrated as follows:

$$\begin{array}{ll} P_1 : \boxed{x_1} & P_1 : \boxed{x_1 + x_2 + \cdots + x_p} \\ P_2 : \boxed{x_2} & P_2 : \boxed{x_2} \\ \vdots \quad \overset{accumulation}{\Longrightarrow} & \vdots \\ P_p : \boxed{x_p} & P_p : \boxed{x_p} \end{array}$$

  The addition is used as reduction operation. To perform a single-accumulation, each processor explicitly calls the operation and specifies the rank of the root processor, the reduction operation to be applied, and the local data block provided. The root processor additionally specifies the buffer in which the accumulated result should be stored.
- **Gather**: For a gather operation, each processor provides a data block, and the data blocks of all processors are collected at a specific root processor P_i. No reduction operation is applied, i.e., processor P_i gets p messages. For root processor P_1, the effect of the operation can be illustrated as follows:

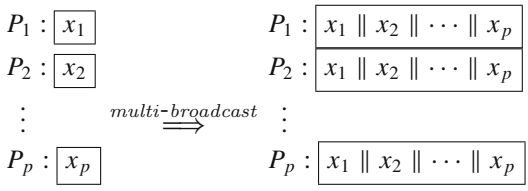
$$\begin{array}{ll} P_1 : \boxed{x_1} & P_1 : \boxed{x_1 \parallel x_2 \parallel \cdots \parallel x_p} \\ P_2 : \boxed{x_2} & P_2 : \boxed{x_2} \\ \vdots \quad \overset{gather}{\Longrightarrow} & \vdots \\ P_p : \boxed{x_p} & P_p : \boxed{x_p} \end{array}$$

  Here, the symbol || denotes the concatenation of the received data blocks. To perform the gather, each processor explicitly calls a gather operation and specifies the local data block provided as well as the rank of the root processor. The root processor additionally specifies a receive buffer in which all data blocks are collected. This buffer must be large enough to store all blocks. After the operation is completed, the receive buffer of the root processor contains the data blocks of all processors in rank order.
- **Scatter**: For a scatter operation, a specific root processor P_i provides a separate data block for every other processor. For root processor P_1, the effect of the operation can be illustrated as follows:



To perform the scatter, each processor explicitly calls a scatter operation and specifies the root processor as well as a receive buffer. The root processor additionally specifies a send buffer in which the data blocks to be sent are provided in rank order of the rank $i = 1, \ldots, p$.

- **Multi-broadcast**: The effect of a multi-broadcast operation is the same as the execution of several single-broadcast operations, one for each processor, i.e., each processor sends the *same* data block to every other processor. From the receiver's point of view, each processor receives a data block from every other processor. Different receivers get the same data block from the same sender. The operation can be illustrated as follows:



In contrast to the global operations considered so far, there is *no* root processor. To perform the multi-broadcast, each processor explicitly calls a multi-broadcast operation and specifies a send buffer which contains the data block as well as a receive buffer. After the completion of the operation, the receive buffer of every processor contains the data blocks provided by all processors in rank order, including its own data block. Multi-broadcast operations are useful to collect blocks of an array that have been computed in a distributed way and to make the entire array available to all processors.

- **Multi-accumulation**: The effect of a multi-accumulation operation is that each processor executes a single-accumulation operation, i.e., each processor provides for every other processor a potentially different data block. The data blocks for the same receiver are combined with a given reduction operation such that *one* (reduced) data block arrives at the receiver. There is no root processor, since each processor acts as a receiver for one accumulation operation. The effect of the operation with addition as reduction operation can be illustrated as follows:

The data block provided by processor P_i for processor P_j is denoted as x_{ij}, $i, j = 1, \ldots, p$. To perform the multi-accumulation, each processor explicitly calls a multi-accumulation operation and specifies a send buffer, a receive buffer, and a reduction operation. In the send buffer, each processor provides a separate data block for each other processor, stored in rank order. After the completion of the operation, the receive buffer of each processor contains the accumulated result for this processor.

- **Total exchange**: For a total exchange operation, each processor provides for each other processor a potentially different data block. These data blocks are sent to their intended receivers, i.e., each processor executes a scatter operation. From a receiver's point of view, each processor receives a data block from each other processor. In contrast to a multi-broadcast, different receivers get different data blocks from the same sender. There is no root processor. The effect of the operation can be illustrated as follows:

$$
\begin{array}{lll}
P_1: & \boxed{x_{11} \parallel x_{12} \parallel \cdots \parallel x_{1p}} & \\
P_2: & \boxed{x_{21} \parallel x_{22} \parallel \cdots \parallel x_{2p}} & \\
\vdots & & \\
P_p: & \boxed{x_{p1} \parallel x_{p2} \parallel \cdots \parallel x_{pp}} &
\end{array}
\quad \overset{total\ exchange}{\Longrightarrow} \quad
\begin{array}{ll}
P_1: & \boxed{x_{11} \parallel x_{21} \parallel \cdots \parallel x_{p1}} \\
P_2: & \boxed{x_{12} \parallel x_{22} \parallel \cdots \parallel x_{p2}} \\
\vdots & \\
P_p: & \boxed{x_{1p} \parallel x_{2p} \parallel \cdots \parallel x_{pp}}
\end{array}
$$

To perform the total exchange, each processor specifies a send buffer and a receive buffer. The send buffer contains the data blocks provided for the other processors in rank order. After the completion of the operation, the receive buffer of each processor contains the data blocks gathered from the other processors in rank order.

Section 4.3.1 considers the implementation of these global communication operations for different networks and derives running times. Chapter 5 describes how these communication operations are provided by the MPI library.

3.5.2.2 Duality of Communication Operations

A single-broadcast operation can be implemented by using a *spanning tree* with the sending processor as root. Edges in the tree correspond to physical connections in the underlying interconnection network. Using a graph representation $G = (V, E)$ of the network, see Sect. 2.5.2, a spanning tree can be defined as a subgraph $G' = (V, E')$ which contains all nodes of V and a subset $E' \subseteq E$ of the edges such that E' represents a tree. The construction of a spanning tree for different networks is considered in Sect. 4.3.1.

Given a spanning tree, a single-broadcast operation can be performed by a top-down traversal of the tree such that starting from the root each node forwards the message to be sent to its children as soon as the message arrives. The message can be forwarded over different links at the same time. For the forwarding, the tree edges can be partitioned into stages such that the message can be forwarded concurrently

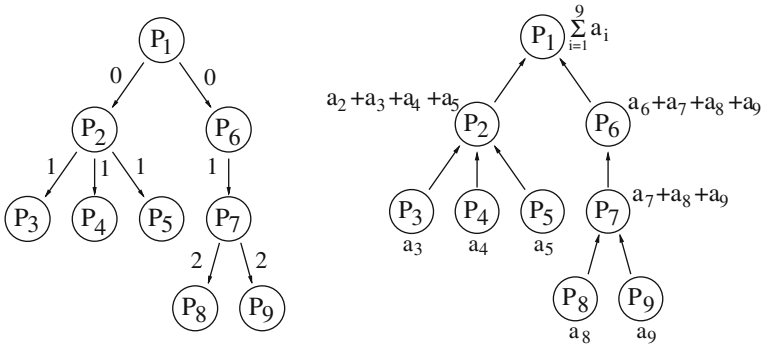


Fig. 3.8 Implementation of a single-broadcast operation using a spanning tree (*left*). The edges of the tree are annotated with the *stage number*. The *right* tree illustrates the implementation of a single-accumulation with the same spanning tree. Processor P_i provides a value a_i for $i = 1, \ldots, 9$. The result is accumulated at the root processor P_1 [19]

over all edges of a stage. Figure 3.8 (left) shows a spanning tree with root P_1 and three stages 0, 1, 2.

Similar to a single-broadcast, a single-accumulation operation can also be implemented by using a spanning tree with the accumulating processor as root. The reduction is performed at the inner nodes according to the given reduction operation. The accumulation results from a bottom-up traversal of the tree, see Fig. 3.8 (right). Each node of the spanning tree receives a data block from each of its children (if present), combines these blocks according to the given reduction operation, including its own data block, and forwards the results to its parent node. Thus, one data block is sent over each edge of the spanning tree, but in the opposite direction as has been done for a single-broadcast. Since the same spanning trees can be used, single-broadcast and single-accumulation are *dual* operations.

A duality relation also exists between a gather and a scatter operation as well as between a multi-broadcast and a multi-accumulation operation.

A scatter operation can be implemented by a top-down traversal of a spanning tree where each node (except the root) receives a set of data blocks from its parent node and forwards those data blocks that are meant for a node in a subtree to its corresponding child node being the root of that subtree. Thus, the number of data blocks forwarded over the tree edges decreases on the way from the root to the leaves. Similarly, a gather operation can be implemented by a bottom-up traversal of the spanning tree where each node receives a set of data blocks from each of its child nodes (if present) and forwards all data blocks received, including its own data block, to its parent node. Thus, the number of data blocks forwarded over the tree edges increases on the way from the leaves to the root. On each path to the root, over each tree edge the same number of data blocks are sent as for a scatter operation, but in opposite direction. Therefore, gather and scatter are dual operations. A multi-broadcast operation can be implemented by using p spanning trees where each spanning tree has a different root processor. Depending on the

underlying network, there may or may not be physical network links that are used multiple times in different spanning trees. If no links are shared, a transfer can be performed concurrently over all spanning trees without waiting, see Sect. 4.3.1 for the construction of such sets of spanning trees for different networks. Similarly, a multi-accumulation can also be performed by using p spanning trees, but compared to a multi-broadcast, the transfer direction is reversed. Thus, multi-broadcast and multi-accumulation are also dual operations.

3.5.2.3 Hierarchy of Communication Operations

The communication operations described form a hierarchy in the following way: Starting from the most general communication operation (total exchange), the other communication operations result by a stepwise *specialization*. A total exchange is the most general communication operation, since each processor sends a potentially *different* message to each other processor. A multi-broadcast is a special case of a total exchange in which each processor sends the *same* message to each other, i.e., instead of p different messages, each processor provides only one message. A multi-accumulation is also a special case of a total exchange for which the messages arriving at an intermediate node are combined according to the given reduction operation before they are forwarded. A gather operation with root P_i is a special case of a multi-broadcast which results from considering only one of the receiving processors, P_i, which receives a message from every other processor. A scatter operation with root P_i is a special case of multi-accumulation which results by using a special reduction operation which forwards the messages of P_i and ignores all other messages. A single-broadcast is a special case of a scatter operation in

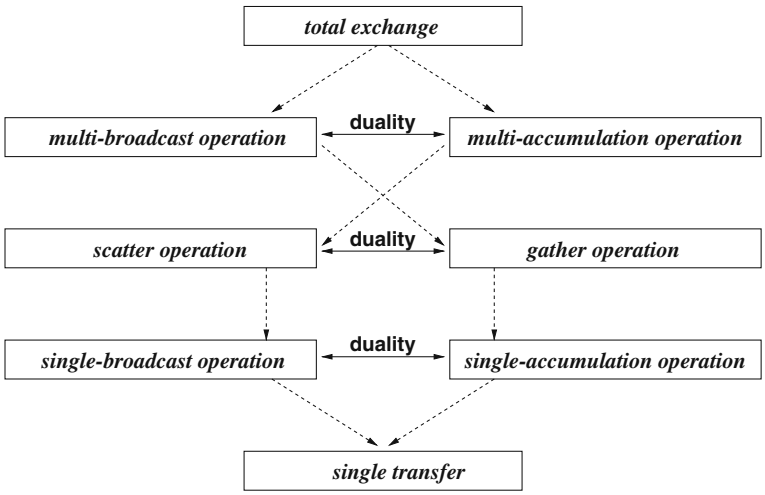


Fig. 3.9 Hierarchy of global communication operations. The *horizontal arrows* denote duality relations. The *dashed arrows* show specialization relations [19]

which the root processor sends the *same* message to every other processor, i.e., instead of p different messages the root processor provides only one message. A single-accumulation is a special case of a gather operation in which a reduction is performed at intermediate nodes of the spanning tree such that only *one* (combined) message results at the root processor. A single transfer between processors P_i and P_j is a special case of a single-broadcast with root P_i for which only the path from P_i to P_j is relevant. A single transfer is also a special case of a single-accumulation with root P_j using a special reduction operation which forwards only the message from P_i. In summary, the hierarchy in Fig. 3.9 results.

3.6 Parallel Matrix–Vector Product

The matrix–vector multiplication is a frequently used component in scientific computing. It computes the product $\mathbf{Ab} = \mathbf{c}$, where $\mathbf{A} \in \mathbb{R}^{n \times m}$ is an $n \times m$ matrix and $\mathbf{b} \in \mathbb{R}^m$ is a vector of size m. (In this section, we use bold-faced type for the notation of matrices or vectors and normal type for scalar values.) The sequential computation of the matrix–vector product

$$c_i = \sum_{j=1}^{m} a_{ij} b_j, \quad i = 1, ..., n,$$

with $\mathbf{c} = (c_1, \ldots, c_n) \in \mathbb{R}^n$, $\mathbf{A} = (a_{ij})_{i=1,\ldots,n,\, j=1,\ldots,m}$, and $\mathbf{b} = (b_1, \ldots, b_m)$, can be implemented in two ways, differing in the loop order of the loops over i and j. First, the matrix–vector product is considered as the computation of n scalar products between rows $\mathbf{a}_1, \ldots, \mathbf{a}_n$ of $\mathbf{A}$ and vector $\mathbf{b}$, i.e.,

$$\mathbf{A} \cdot \mathbf{b} = \begin{pmatrix} (\mathbf{a}_1, \mathbf{b}) \\ \vdots \\ (\mathbf{a}_n, \mathbf{b}) \end{pmatrix},$$

where $(\mathbf{x}, \mathbf{y}) = \sum_{j=1}^{m} x_j y_j$ for $\mathbf{x}, \mathbf{y} \in \mathbb{R}^m$ with $\mathbf{x} = (x_1, \ldots, x_m)$ and $\mathbf{y} = (y_1, \ldots, y_m)$ denotes the scalar product (or inner product) of two vectors. The corresponding algorithm (in C notation) is

```
for (i=0; i<n; i++) c[i] = 0;
for (i=0; i<n; i++)
    for (j=0; j<m; j++)
        c[i] = c[i] + A[i][j] * b[j];
```

The matrix $\mathbf{A} \in \mathbb{R}^{n \times m}$ is implemented as a two-dimensional array `A` and the vectors $\mathbf{b} \in \mathbb{R}^m$ and $\mathbf{c} \in \mathbb{R}^n$ are implemented as one-dimensional arrays `b` and `c`. (The indices start with 0 as usual in C.) For each `i = 0,...,n-1`, the inner loop body consists of a loop over `j` computing one of the scalar products. Second, the

matrix–vector product can be written as a linear combination of columns $\tilde{\mathbf{a}}_1, \ldots, \tilde{\mathbf{a}}_m$ of $\mathbf{A}$ with coefficients $b_1, \ldots, b_m$, i.e.,

$$\mathbf{A} \cdot \mathbf{b} = \sum_{j=1}^{m} b_j \tilde{\mathbf{a}}_j \ .$$

The corresponding algorithm (in C notation) is:

```
for (i=0; i<n; i++) c[i] = 0;
for (j=0; j<m; j++)
    for (i=0; i<n; i++)
        c[i] = c[i] + A[i][j] * b[j] ;
```

For each `j = 0,...,m-1`, a column $\tilde{\mathbf{a}}_j$ is added to the linear combination. Both sequential programs are equivalent since there are no dependencies and the loops over i and j can be exchanged. For a parallel implementation, the row- and column-oriented representations of matrix $\mathbf{A}$ give rise to different parallel implementation strategies.

(a) The row-oriented representation of matrix $\mathbf{A}$ in the computation of n scalar products $(\mathbf{a}_i, \mathbf{b})$, $i = 1, \ldots, n$, of rows of $\mathbf{A}$ with vector $\mathbf{b}$ leads to a parallel implementation in which each processor of a set of p processors computes approximately n/p scalar products.
(b) The column-oriented representation of matrix $\mathbf{A}$ in the computation of the linear combination $\sum_{j=1}^{m} b_j \tilde{\mathbf{a}}_j$ of columns of $\mathbf{A}$ leads to a parallel implementation in which each processor computes a part of this linear combination with approximately m/p column vectors.

In the following, we consider these parallel implementation strategies for the case of n and m being multiples of the number of processors p.

3.6.1 Parallel Computation of Scalar Products

For a parallel implementation of a matrix–vector product on a distributed memory machine, the data distribution of $\mathbf{A}$ and $\mathbf{b}$ is chosen such that the processor computing the scalar product $(\mathbf{a}_i, \mathbf{b})$, $i \in \{1, \ldots, n\}$, accesses only data elements stored in its private memory, i.e., row $\mathbf{a}_i$ of $\mathbf{A}$ and vector $\mathbf{b}$ are stored in the private memory of the processor computing the corresponding scalar product. Since vector $\mathbf{b} \in \mathbb{R}^m$ is needed for all scalar products, $\mathbf{b}$ is stored in a replicated way. For matrix $\mathbf{A}$, a row-oriented data distribution is chosen such that a processor computes the scalar product for which the matrix row can be accessed locally. Row-oriented blockwise as well as cyclic or block–cyclic data distributions can be used.

For the row-oriented blockwise data distribution of matrix $\mathbf{A}$, processor P_k, $k = 1, \ldots, p$, stores the rows $\mathbf{a}_i$, $i = n/p \cdot (k-1) + 1, \ldots, n/p \cdot k$, in its private memory and computes the scalar products $(\mathbf{a}_i, \mathbf{b})$. The computation of $(\mathbf{a}_i, \mathbf{b})$ needs

no data from other processors and, thus, no communication is required. According to the row-oriented blockwise computation the result vector $\mathbf{c} = (c_1, \ldots, c_n)$ has a blockwise distribution.

When the matrix–vector product is used within a larger algorithm like iteration methods, there are usually certain requirements for the distribution of **c**. In iteration methods, there is often the requirement that the result vector **c** has the same data distribution as the vector **b**. To achieve a replicated distribution for **c**, each processor P_k, $k = 1, \ldots, p$, sends its block $(c_{n/p\cdot(k-1)+1}, \ldots, c_{n/p\cdot k})$ to all other processors. This can be done by a multi-broadcast operation. A parallel implementation of the matrix–vector product including this communication is given in Fig. 3.10. The program is executed by all processors P_k, $k = 1, \ldots, p$, in the SPMD style. The communication operation includes an implicit barrier synchronization. Each processor P_k stores a different part of the $n \times m$ array `A` in its local array `local_A` of dimension `local_n` $\times$ `m`. The block of rows stored by P_k in `local_A` contains the global elements

```
local_A[i][j]=A[i+(k-1) * n/p][j]
```

with $i = 0, \ldots, n/p - 1$, $j = 0, \ldots, m - 1$, and $k = 1, \ldots, p$. Each processor computes a local matrix–vector product of array `local_A` with array `b` and stores the result in array `local_c` of size `local_n`. The communication operation

```
multi_broadcast(local_c,local_n,c)
```

performs a multi-broadcast operation with the local arrays `local_c` of all processors as input. After this communication operation, the global array `c` contains the values

```
c[i+(k-1) * n/p]=local_c[i]
```

for $i = 0, \ldots, n/p - 1$ and $k = 1, \ldots, p$, i.e., the array `c` contains the values of the local vectors in the order of the processors and has a replicated data distribution.

```
/* Matrix-vector product Ab = c with parallel inner products*/
/* Row-oriented blockwise distribution of A */
/* Replicated distribution of vectors b and c */
local_n = n/p;
for (i=0; i<local_n; i++) local_c[i] = 0;
for (i=0; i<local_n; i++)
    for (j=0; j<m; j++)
        local_c[i] = local_c[i] + local_A[i][j] * b[j];
multi_broadcast(local_c,local_n,c);
/* Multi-broadcast operation of (c[0],...,c[local_n]) to global_c*/
```

Fig. 3.10 Program fragment in C notation for a parallel program of the matrix–vector product with row-oriented blockwise distribution of the matrix **A** and a final redistribution of the result vector **c**

See Fig. 3.13(1) for an illustration of the data distribution of A, b, and c for the program given in Fig. 3.10.

For a row-oriented cyclic distribution, each processor P_k, $k = 1, \ldots, p$, stores the rows $\mathbf{a}_i$ of matrix $\mathbf{A}$ with $i = k + p \cdot (l - 1)$ for $l = 1, \ldots, n/p$ and computes the corresponding scalar products. The rows in the private memory of processor P_k are stored within one local array local_A of dimension local_n × m. After the parallel computation of the result array local_c, the entries have to be reordered correspondingly to get the global result vector in the original order.

For the implementation of the matrix–vector product on a **shared memory** machine, the row-oriented distribution of the matrix $\mathbf{A}$ and the corresponding distribution of the computation can be used. Each processor of the shared memory machine computes a set of scalar products as described above. A processor P_k computes n/p elements of the result vector $\mathbf{c}$ and uses n/p corresponding rows of matrix $\mathbf{A}$ in a blockwise or cyclic way, $k = 1, \ldots, p$. The difference to the implementation on a distributed memory machine is that an explicit distribution of the data is not necessary since the entire matrix $\mathbf{A}$ and vector $\mathbf{b}$ reside in the common memory accessible by all processors.

The distribution of the computation to processors according to a row-oriented distribution, however, causes the processors to access different elements of $\mathbf{A}$ and compute different elements of $\mathbf{c}$. Thus, the write accesses to $\mathbf{c}$ cause no conflict. Since the accesses to matrix $\mathbf{A}$ and vector $\mathbf{b}$ are read accesses, they also cause no conflict. Synchronization and locking are not required for this shared memory implementation. Figure 3.11 shows an SPMD program for a parallel matrix–vector multiplication accessing the global arrays A, b, and c. The variable k denotes the processor id of the processor P_k, $k = 1, \ldots, p$. Because of this processor number k, each processor P_k computes different elements of the result array c. The program fragment ends with a barrier synchronization synch() to guarantee that all processors reach this program point and the entire array c is computed before any processor executes subsequent program parts. (The same program can be used for a distributed memory machine when the entire arrays A, b, and c are allocated in each private memory; this approach needs much more memory since the arrays are allocated p times.)

```
/* Matrix-vector product Ab=c with parallel inner products*/
/* Row-oriented distribution of the computation */
local_n = n/p;
for (i=0; i<local_n; i++) c[i+(k-1)*local_n] = 0;
for (i=0; i<local_n; i++)
    for (j=0; j<m; j++)
        c[i+(k-1)*local_n] =
        c[i+(k-1)*local_n] + A[i+(k-1)*local_n][j] * b[j];
synch();
```

Fig. 3.11 Program fragment in C notation for a parallel program of the matrix–vector product with row-oriented blockwise distribution of the computation. In contrast to the program in Fig. 3.10, the program uses the global arrays A, b, and c for a shared memory system

3.6.2 Parallel Computation of the Linear Combinations

For a distributed memory machine, the parallel implementation of the matrix–vector product in the form of the linear combination uses a **column-oriented** distribution of the matrix **A**. Each processor computes the part of the linear combination for which it owns the corresponding columns $\tilde{\mathbf{a}}_i$, $i \in \{1, \ldots, m\}$. For a blockwise distribution of the columns of **A**, processor P_k owns the columns $\tilde{\mathbf{a}}_i$, $i = m/p \cdot (k-1) + 1, \ldots, m/p \cdot k$, and computes the n-dimensional vector

$$\mathbf{d}_k = \sum_{j=m/p\cdot(k-1)+1}^{m/p\cdot k} b_j \tilde{\mathbf{a}}_j,$$

which is a partial linear combination and a part of the total result, $k = 1, \ldots, p$. For this computation only a block of elements of vector **b** is accessed and only this block needs to be stored in the private memory. After the parallel computation of the vectors $\mathbf{d}_k$, $k = 1, \ldots, p$, these vectors are added to give the final result $\mathbf{c} = \sum_{k=1}^{p} \mathbf{d}_k$. Since the vectors $\mathbf{d}_k$ are stored in different local memories, this addition requires communication, which can be performed by an accumulation operation with the addition as reduction operation. Each of the processors P_k provides its vector $\mathbf{d}_k$ for the accumulation operation. The result of the accumulation is available on one of the processors. When the vector is needed in a replicated distribution, a broadcast operation is performed. The data distribution before and after the communication is illustrated in Fig. 3.13(2a). A parallel program in the SPMD style is given in Fig. 3.12. The local arrays `local_b` and `local_A` store blocks of **b** and blocks of columns of **A** so that each processor P_k owns the elements

```
local_A[i][j]=A[i][j+(k-1) * m/p]
```

and

```
local_b[j]=b[j+(k-1) * m/p],
```

```
/* Matrix-vector product Ab=c with parallel linear combination*/
/* Column-oriented distribution of A */
/* Replicated distribution of vectors b and c */
local_m=m/p;
for (i=0; i<n; i++) d[i] = 0;
for (j=0; j<local_m; j++)
    for (i=0 ;i<n; i++)
        d[i] = d[i] + local_b[j] * local_A[i][j];
single_accumulation(d,n,c,ADD,1);
single_broadcast(c,1);
```

Fig. 3.12 Program fragment in C notation for a parallel program of the matrix–vector product with column-oriented blockwise distribution of the matrix **A** and reduction operation to compute the result vector **c**. The program uses local array `d` for the parallel computation of partial linear combinations

where `j=0,...,m/p-1`, `i=0,...,n-1`, and `k=1,...,p`. The array `d` is a private vector allocated by each of the processors in its private memory containing different data after the computation. The operation

```
single_accumulation(d,local_m,c,ADD,1)
```

denotes an accumulation operation, for which each processor provides its array `d` of size n, and `ADD` denotes the reduction operation. The last parameter is 1 and means that processor P_1 is the root processor of the operation, which stores the result of the addition into the array `c` of length n. The final `single_broadcast(c,1)` sends the array `c` from processor P_1 to all other processors and a replicated distribution of `c` results.

Alternatively to this final communication, multi-accumulation operation can be applied which leads to a blockwise distribution of array `c`. This program version may be advantageous if `c` is required to have the same distribution as array `b`. Each processor accumulates the n/p elements of the local arrays `d`, i.e., each processor computes a block of the result vector `c` and stores it in its local memory. This communication is illustrated in Fig. 3.13(2b).

For shared memory machines, the parallel computation of the linear combinations can also be used but special care is needed to avoid access conflicts for the write accesses when computing the partial linear combinations. To avoid write conflicts, a separate array `d_k` of length n should be allocated for each of the processors P_k to compute the partial result in parallel without conflicts. The final accumulation needs no communication, since the data `d_k` are in the common memory, and can be performed in a blocked way.

The computation and communication time for the matrix–vector product is analyzed in Sect. 4.4.2.

3.7 Processes and Threads

Parallel programming models are often based on processors or threads. Both are abstractions for a flow of control, but there are some differences which we will consider in this section in more detail. As described in Sect. 3.2, the principal idea is to decompose the computation of an application into tasks and to employ multiple control flows running on different processors or cores for their execution, thus obtaining a smaller overall execution time by parallel processing.

3.7.1 Processes

In general, a process is defined as a program in execution. The process comprises the executable program along with all information that is necessary for the execution of the program. This includes the program data on the runtime stack or the heap,

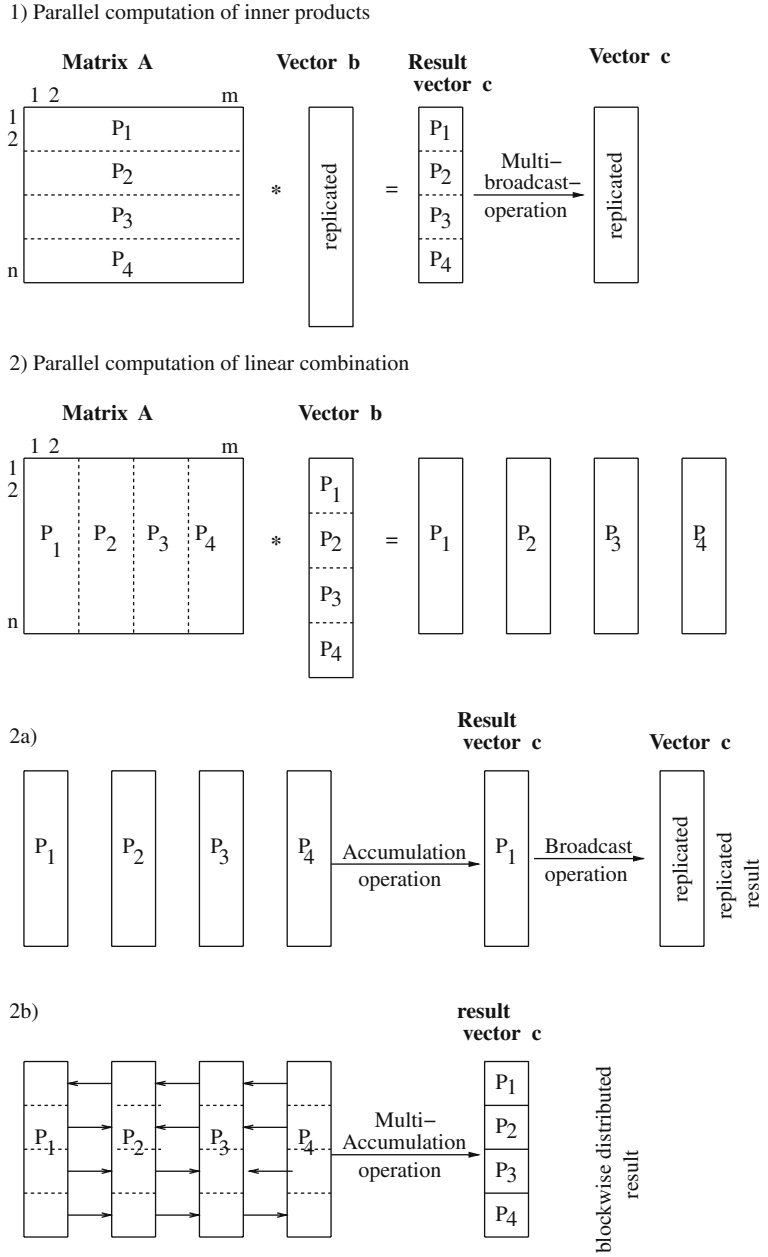


Fig. 3.13 Parallel matrix–vector multiplication with (**1**) parallel computation of scalar products and replicated result and (**2**) parallel computation of linear combinations with (**a**) replicated result and (**b**) blockwise distribution of the result

the current values of the registers, as well as the content of the program counter which specifies the next instruction to be executed. All this information changes dynamically during the execution of the process. Each process has its own address space, i.e., the process has exclusive access to its data. When two processes want to exchange data, this has to be done by explicit communication.

A process is assigned to execution resources (processors or cores) for execution. There may be more processes than execution resources. To bring all processes to execution from time to time, an execution resource typically executes several processes at different points in time, e.g., in a round-robin fashion. If the execution is assigned to another process by the scheduler of the operating system, the state of the suspended process must be saved to allow a continuation of the execution at a later time with the process state before suspension. This switching between processes is called **context switch**, and it may cause a significant overhead, depending on the hardware support [137]. Often time slicing is used to switch between the processes. If there is a single execution resource only, the active processes are executed concurrently in a time-sliced way, but there is no real parallelism. If several execution resources are available, different processes can be executed by different execution resources, thus indeed leading to a parallel execution.

When a process is generated, it must obtain the data required for its execution. In Unix systems, a process P_1 can create a new process P_2 with the `fork` system call. The new child process P_2 is an identical copy of the parent process P_1 at the time of the `fork` call. This means that the child process P_2 works on a *copy* of the address space of the parent process P_1 and executes the same program as P_1, starting with the instruction following the `fork` call. The child process gets its own process number and, depending on this process number, it can execute different statements as the parent process. Since each process has its own address space and since process creation includes the generation of a copy of the address space of the parent process, process creation and management may be quite time-consuming. Data exchange between processes is often done via socket communication which is based on TCP/IP or UDP/IP communication. This may lead to a significant overhead, depending on the socket implementation and the speed of the interconnection between the execution resources assigned to the communicating processes.

3.7.2 Threads

The thread model is an extension of the process model. In the thread model, each process may consist of *multiple* independent control flows which are called **threads**. The word *thread* is used to indicate that a potentially long continuous sequence of instructions is executed. During the execution of a process, the different threads of this process are assigned to execution resources by a scheduling method.

3.7.2.1 Basic Concepts of Threads

A significant feature of threads is that threads of *one* process share the address space of the process, i.e., they have a common address space. When a thread stores a value

in the shared address space, another thread of the same process can access this value afterwards. Threads are typically used if the execution resources used have access to a physically shared memory, as is the case for the cores of a multicore processor. In this case, information exchange is fast compared to socket communication. Thread generation is usually much faster than process generation: No copy of the address space is necessary since the threads of a process share the address space. Therefore, the use of threads is often more flexible than the use of processes, yet providing the same advantages concerning a parallel execution. In particular, the different threads of a process can be assigned to different cores of a multicore processor, thus providing parallelism within the processes.

Threads can be provided by the runtime system as **user-level threads** or by the operating system as **kernel threads**. User-level threads are managed by a *thread library* without specific support by the operating system. This has the advantage that a switch from one thread to another can be done without interaction of the operating system and is therefore quite fast. Disadvantages of the management of threads at user level come from the fact that the operating system has no knowledge about the existence of threads and manages entire processes only. Therefore, the operating system cannot map different threads of the same process to different execution resources and all threads of one process are executed on the same execution resource. Moreover, the operating system cannot switch to another thread if one thread executes a blocking I/O operation. Instead, the CPU scheduler of the operating system suspends the entire process and assigns the execution resource to another process.

These disadvantages can be avoided by using kernel threads, since the operating system is aware of the existence of threads and can react correspondingly. This is especially important for an efficient use of the cores of a multicore system. Most operating systems support threads at the kernel level.

3.7.2.2 Execution Models for Threads

If there is no support for thread management by the operating system, the thread library is responsible for the entire thread scheduling. In this case, *all* user-level threads of a user process are mapped to *one* process of the operating system. This is called ***N*:1 mapping**, or *many-to-one mapping*, see Fig. 3.14 for an illustration. At each point in time, the library scheduler determines which of the different threads comes to execution. The mapping of the processes to the execution resources is done by the operating system. If several execution resources are available, the operating system can bring several processes to execution concurrently, thus exploiting parallelism. But with this organization the execution of different threads of one process on different execution resources is not possible.

If the operating system supports thread management, there are two possibilities for the mapping of user-level threads to kernel threads. The first possibility is to generate a kernel thread for each user-level thread. This is called **1:1 mapping**, or *one-to-one mapping*, see Fig. 3.15 for an illustration. The scheduler of the operating system selects which kernel threads are executed at which point in time. If

This figure will be printed in b/w

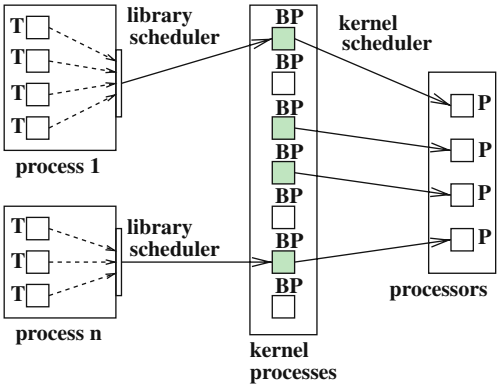
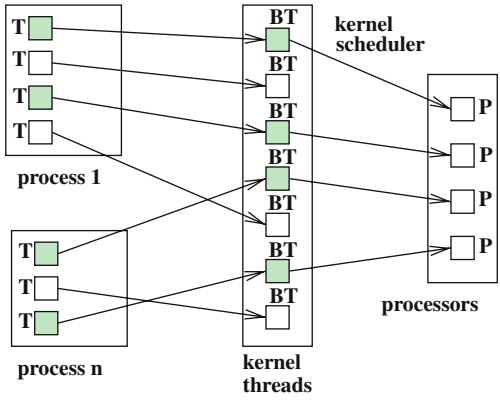


Fig. 3.14 Illustration of a N:1 mapping for thread management without kernel threads. The scheduler of the thread library selects the next thread T of the user process for execution. Each user process is assigned to exactly one process BP of the operating system. The scheduler of the operating system selects the processes to be executed at a certain time and maps them to the execution resources P

This figure will be printed in b/w

Fig. 3.15 Illustration of a 1:1 mapping for thread management with kernel threads. Each user-level thread T is assigned to one kernel thread BT. The kernel threads BT are mapped to execution resources P by the scheduler of the operating system



multiple execution resources are available, it also determines the mapping of the kernel threads to the execution resources. Since each user-level thread is assigned to exactly one kernel thread, there is no need for a library scheduler. Using a 1:1 mapping, different threads of a user process can be mapped to different execution resources, if enough resources are available, thus leading to a parallel execution within a single process.

The second possibility is to use a two-level scheduling where the scheduler of the thread library assigns the user-level threads to a given set of kernel threads. The scheduler of the operating system maps the kernel threads to the available execution resources. This is called ***N*:*M* mapping**, or *many-to-many mapping*, see Fig. 3.16 for an illustration. At different points in time, a user thread may be mapped to a different kernel thread, i.e., no fixed mapping is used. Correspondingly, at different

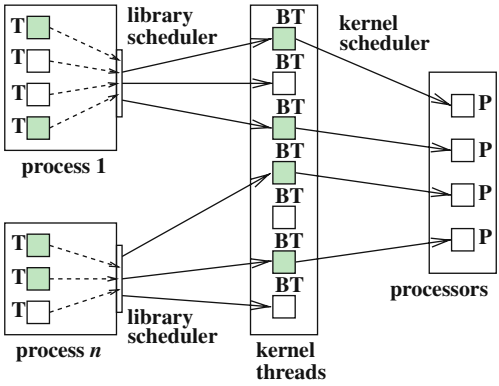


Fig. 3.16 Illustration of an $N{:}M$ mapping for thread management with kernel threads using a two-level scheduling. User-level threads T of different processes are assigned to a set of kernel threads BT ($N{:}M$ mapping) which are then mapped by the scheduler of the operating system to execution resources P

This figure will be printed in b/w

points in time, a kernel thread may execute different user threads. Depending on the thread library, the programmer can influence the scheduler of the library, e.g., by selecting a scheduling method as is the case for the Pthreads library, see Sect. 6.1.10 for more details. The scheduler of the operating system on the other hand is tuned for an efficient use of the hardware resources, and there is typically no possibility for the programmer to directly influence the behavior of this scheduler. This second mapping possibility usually provides more flexibility than a 1:1 mapping, since the programmer can adapt the number of user-level threads to the specific algorithm or application. The operating system can select the number of kernel threads such that an efficient management and mapping of the execution resources is facilitated.

3.7.2.3 Thread States

A thread can be in one of the following states:

- **newly generated**, i.e., the thread has just been generated, but has not yet performed any operation;
- **executable**, i.e., the thread is ready for execution, but is currently not assigned to any execution resources;
- **running**, i.e., the thread is currently being executed by an execution resource;
- **waiting**, i.e., the thread is waiting for an external event to occur; the thread cannot be executed before the external event happens;
- **finished**, i.e., the thread has terminated all its operations.

Figure 3.17 illustrates the transition between these states. The transitions between the states *executable* and *running* are determined by the scheduler. A thread may enter the state *waiting* because of a blocking I/O operation or because of the execution of a synchronization operation which causes it to be blocked. The transition from the state *waiting* to *executable* may be caused by a termination of a previously issued I/O operation or because another thread releases the resource which this thread is waiting for.

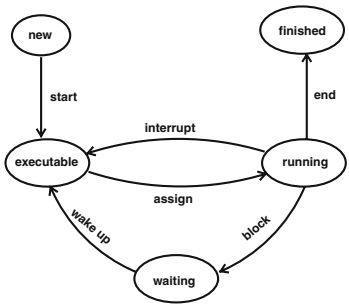


Fig. 3.17 States of a thread. The *nodes* of the diagram show the possible states of a thread and the *arrows* show possible transitions between them

3.7.2.4 Visibility of Data

The different threads of a process share a common address space. This means that the global variables of a program and all dynamically allocated data objects can be accessed by any thread of this process, no matter which of the threads has allocated the object. But for each thread, there is a private runtime stack for controlling function calls of this thread and to store the local variables of these functions, see Fig. 3.18 for an illustration. The data kept on the runtime stack is local data of the corresponding thread and the other threads have no direct access to this data. It is in principle possible to give them access by passing an address, but this is dangerous, since how long the data is accessible cannot be predicted. The stack frame of a function call is freed as soon as the function call is terminated. The runtime stack of a thread exists only as long as the thread is *active*; it is freed as soon as the thread is terminated. Therefore, a return value of a thread should not be passed via its runtime stack. Instead, a global variable or a dynamically allocated data object should be used, see Chap. 6 for more details.

This figure will be printed in b/w

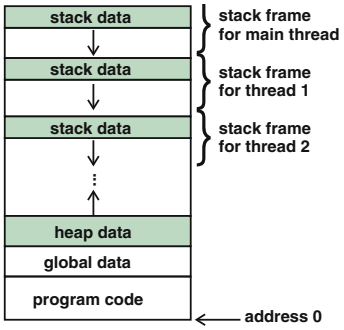


Fig. 3.18 Runtime stack for the management of a program with multiple threads

3.7.3 Synchronization Mechanisms

When multiple threads execute a parallel program in parallel, their execution has to be coordinated to avoid race conditions. Synchronization mechanisms are provided

to enable a coordination, e.g., to ensure a certain execution order of the threads or to control access to shared data structures. Synchronization for shared variables is mainly used to avoid a concurrent manipulation of the same variable by different threads, which may lead to non-deterministic behavior. This is important for multi-threaded programs, no matter whether a single execution resource is used in a time-slicing way or whether several execution resources execute multiple threads in parallel. Different synchronization mechanisms are provided for different situations. In the following, we give a short overview.

3.7.3.1 Lock Synchronization

For a concurrent access of shared variables, race conditions can be avoided by a **lock mechanism** based on predefined **lock variables**, which are also called **mutex variables** as they help to ensure mutual exclusion. A lock variable `l` can be in one of two states: *locked* or *unlocked*. Two operations are provided to influence this state: `lock(l)` and `unlock(l)`. The execution of `lock(l)` locks `l` such that it cannot be locked by another thread; after the execution, `l` is in the *locked* state and the thread that has executed `lock(l)` is the *owner* of `l`. The execution of `unlock(l)` unlocks a previously locked lock variable `l`; after the execution, `l` is in the *unlocked* state and has no owner. To avoid race conditions for the execution of a program part, a lock variable `l` is assigned to this program part and each thread executes `lock(l)` before entering the program part and `unlock(l)` after leaving the program part. To avoid race conditions, *each* of the threads must obey this programming rule.

A call of `lock(l)` for a lock variable `l` has the effect that the executing thread T_1 becomes the owner of `l`, if `l` has been in the *unlocked* state before. But if there is already another owner T_2 of `l` before T_1 calls `lock(l)`, T_1 is blocked until T_2 has called `unlock(l)` to release `l`. If there are blocked threads waiting for `l` when `unlock(l)` is called, one of the waiting threads is woken up and becomes the new owner of `l`. Thus, using a lock mechanism in the described way leads to a *sequentialization* of the execution of a program part which ensures that at each point in time, only one thread executes the program part. The provision of lock mechanisms in libraries like Pthreads, OpenMP, or Java threads is described in Chap. 6.

It is important to see that mutual exclusion for accessing a shared variable can only be guaranteed if all threads use a lock synchronization to access the shared variable. If this is not the case, a race condition may occur, leading to an incorrect program behavior. This can be illustrated by the following example where two threads T_1 and T_2 access a shared integer variable s which is protected by a lock variable `l` [112]:

Thread T_1	Thread T_2
`lock(l);`	
`s = 1;`	`s = 2;`
`if (s!=1) fire_missile();`	
`unlock(l);`	

In this example, thread T_1 may get interrupted by the scheduler and thread T_2 can set the value of `s` to 2; if T_1 resumes execution, `s` has value 2 and `fire_missile()` is called. For other execution orders, `fire_missile()` will not be called. This non-deterministic behavior can be avoided if T_2 also uses a lock mechanism with `l` to access `s`.

Another mechanism to ensure mutual exclusion is provided by **semaphores** [40]. A semaphore is a data structure which contains an integer counter s and to which two atomic operations $P(s)$ and $V(s)$ can be applied. A *binary semaphore* s can only have values 0 or 1. For a *counting semaphore*, s can have any positive integer value. The operation $P(s)$, also denoted as `wait(s)`, waits until the value of s is larger than 0. When this is the case, the value of s is decreased by 1, and execution can continue with the subsequent instructions. The operation $V(s)$, also denoted as `signal(s)`, increments the value of s by 1. To ensure mutual exclusion for a critical section, the section is protected by a semaphore s in the following form:

```
wait(s)
critical section
signal(s).
```

Different threads may execute operations $P(s)$ or $V(s)$ for a semaphore s to access the critical section. After a thread T_1 has successfully executed the operation `wait(s)` with waiting it can enter the critical section. Every other thread T_2 is blocked when it executes `wait(s)` and can therefore not enter the critical section. When T_1 executes `signal(s)` after leaving the critical section, one of the waiting threads will be woken up and can enter the critical section.

Another concept to ensure mutual exclusion is the concept of **monitors** [90]. A monitor is a language construct which allows the definition of data structures and access operations. These operations are the *only* means by which the data of a monitor can be accessed. The monitor ensures that the access operations are executed with mutual exclusion, i.e., at each point in time, only one thread is allowed to execute any of the access methods provided.

3.7.3.2 Thread Execution Control

To control the execution of multiple threads, barrier synchronization and condition synchronization can be used. A **barrier synchronization** defines a synchronization point where each thread must wait until all other threads have also reached this synchronization point. Thus, none of the threads executes any statement after the synchronization point until all other threads have also arrived at this point. A barrier synchronization also has the effect that it defines a global state of the shared address space in which all operations specified before the synchronization point have been executed. Statements after the synchronization point can be sure that this global state has been established.

Using a **condition synchronization**, a thread T_1 is blocked until a given condition has been established. The condition could, for example, be that a shared variable

contain a specific value or have a specific state like a shared buffer containing at least one entry. The blocked thread T_1 can only be woken up by another thread T_2, e.g., after T_2 has established the condition which T_1 waits for. When T_1 is woken up, it enters the state *executable*, see Sect. 3.7.2.2, and will later be assigned to an execution resource, then entering the state *running*. Thus, after being woken up, T_1 may not be immediately executed, e.g., if not enough execution resources are available. Therefore, although T_2 may have established the condition which T_1 waits for, it is important that T_1 check the condition again as soon as it is running. The reason for this additional check is that in the meantime another thread T_3 may have performed some computations which might have led to the fact that the condition is not fulfilled any more. Condition synchronization can be supported by condition variables. These are for example provided by Pthreads and must be used together with a lock variable to avoid race condition when evaluating the condition, see Sect. 6.1 for more details. A similar mechanism is provided in Java by `wait()` and `notify()`, see Sect. 6.2.3.

3.7.4 Developing Efficient and Correct Thread Programs

Depending on the requirements of an application and the specific implementation by the programmer, synchronization leads to a complicated interaction between the executing threads. This may cause problems like performance degradation by sequentializations, or even deadlocks. This section contains a short discussion of this topic and gives some suggestions about how efficient thread-based programs can be developed.

3.7.4.1 Number of Threads and Sequentialization

Depending on the design and implementation, the runtime of a parallel program based on threads can be quite different. For the design of a parallel program it is important

- to use a suitable number of threads which should be selected according to the degree of parallelism provided by the application and the number of execution resources available and
- to avoid sequentialization by synchronization operations whenever possible.

When synchronization is necessary, e.g., to avoid race conditions, it is important that the resulting critical section which is executed sequentially be made as small as possible to reduce the resulting waiting times.

The creation of threads is necessary to exploit parallel execution. A parallel program should create a sufficiently large number of threads to provide enough work for all cores of an execution platform, thus using the available resources efficiently. But the number of threads created should not be too large to keep the overhead for thread creation, management, and termination small. For a large number of threads, the work per thread may become quite small, giving the thread overhead a significant

portion of the overall execution time. Moreover, many hardware resources, in particular caches, may be shared by the cores, and performance degradations may result if too many threads share the resources; in the case of caches, a degradation of the read/write bandwidth might result.

The threads of a parallel program must be coordinated to ensure a correct behavior. An example is the use of synchronization operations to avoid race conditions. But too many synchronizations may lead to situations where only one or a small number of threads are active while the other threads are waiting because of a synchronization operation. In effect, this may result in a **sequentialization** of the thread execution, and the available parallelism cannot be used. In such situations, increasing the number of threads does not lead to faster program execution, since the new threads are waiting most of the time.

3.7.4.2 Deadlock

Non-deterministic behavior and race conditions can be avoided by synchronization mechanisms like lock synchronization. But the use of locks can lead to **deadlocks**, when program execution comes into a state where each thread waits for an event that can only be caused by another thread, but this thread is also waiting.

Generally, a deadlock occurs for a set of activities, if each of the activities waits for an event that can only be caused by one of the other activities, such that a cycle of mutual waiting occurs. A deadlock may occur in the following example where two threads T_1 and T_2 both use two locks `s1` and `s2`:

Thread T_1	Thread T_2
`lock(s1);`	`lock(s2);`
`lock(s2);`	`lock(s1):`
`do_work();`	`do_work();`
`unlock(s2)`	`unlock(s1)`
`unlock(s1)`	`unlock(s2)`

A deadlock occurs for the following execution order:

- a thread T_1 first tries to set a lock s_1, and then s_2; after having locked s_1 successfully, T_1 is interrupted by the scheduler;
- a thread T_2 first tries to set lock s_2 and then s_1; after having locked s_2 successfully, T_2 waits for the release of s_1.

In this situation, s_1 is locked by T_1 and s_2 by T_2. Both threads T_1 and T_2 wait for the release of the missing lock by the other thread. But this cannot occur, since the other thread is waiting.

It is important to avoid such mutual or cyclic waiting situations, since the program cannot be terminated in such situations. Specific techniques are available to avoid deadlocks in cases where a thread must set multiple locks to proceed. Such techniques are described in Sect. 6.1.2.

3.7.4.3 Memory Access Times and Cache Effects

Memory access times may constitute a significant portion of the execution time of a parallel program. A memory access issued by a program causes a data transfer from the main memory into the cache hierarchy of that core which has issued the memory access. This data transfer is caused by the read and write operations of the cores. Depending on the specific pattern of read and write operations, not only is there a transfer from main memory to the local caches of the cores, but there may also be a transfer between the local caches of the cores. The exact behavior is controlled by hardware, and the programmer has no direct influence on this behavior.

The transfer within the memory hierarchy can be captured by dependencies between the memory accesses issued by different cores. These dependencies can be categorized as read–read dependency, read–write dependency, and write–write dependency. A read–read dependency occurs if two threads running on different cores access the same memory location. If this memory location is stored in the local caches of both cores, both can read the stored values from their cache, and no access to main memory needs to be done. A read–write dependency occurs, if one thread T_1 executes a write into a memory location which is later read by another thread T_2 running on a different core. If the two cores involved do not share a common cache, the memory location that is written by T_1 must be transferred into main memory after the write before T_2 executes its read which then causes a transfer from main memory into the local cache of the core executing T_2. Thus, a read–write dependency consumes memory bandwidth.

A write–write dependency occurs, if two threads T_1 and T_2 running on different cores perform a write into the same memory location in a given order. Assuming that T_1 writes before T_2, a cache coherency protocol, see Sect. 2.7.3, must ensure that the caches of the participating cores are notified when the memory accesses occur. The exact behavior depends on the protocol and the cache implementation as write-through or write-back, see Sect. 2.7.1. In any case, the protocol causes a certain amount of overhead to handle the write–write dependency.

False sharing occurs if two threads T_1 and T_2, running on different cores, access different memory locations that are held in the same cache line. In this case, the same memory operations must be performed as for an access to the same memory locations, since a cache line is the smallest transfer unit in the memory hierarchy. False sharing can lead to a significant amount of memory transfers and to notable performance degradations. It can be avoided by an alignment of variables to cache line boundaries; this is supported by some compilers.

3.8 Further Parallel Programming Approaches

For the programming of parallel architectures, a large number of approaches have been developed during the last years. A first classification of these approaches can be made according to the memory view provided, shared address space or distributed address space, as discussed earlier. In the following, we give a detailed description of

the most popular approaches for both classes. For a distributed address space, MPI is by far the most often used environment, see Chap. 5 for a detailed description. The use of MPI is not restricted to parallel machines with a physically distributed memory organization. It can also be used for parallel architectures with a physically shared address space like multicore architectures. Popular programming approaches for shared address space include Pthreads, Java threads, and OpenMP, see Chap. 6 for a detailed treatment. But besides these popular environments, there are many other interesting approaches aiming at making parallel programming easier by providing the right abstraction. We give a short overview in this section.

The advent of multicore architectures and their use in normal desktop computers has led to an intensifying of the research efforts to develop a simple, yet efficient parallel language. An important argument for the need of such a language is that parallel programming with processes or threads is difficult and is a big step for programmers used to sequential programming [114]. It is often mentioned that, for example, thread programming with lock mechanisms and other forms of synchronization are too low level and too error-prone, since problems like race conditions or deadlocks can easily occur. Current techniques for parallel software development are therefore sometimes compared to assembly programming [169].

In the following, we give a short description of language approaches which attempt to provide suitable mechanisms at the right level of abstraction. Moreover, we give a short introduction to the concept of transactional memory.

3.8.1 Approaches for New Parallel Languages

In this subsection, we give a short overview of interesting approaches for new parallel languages that are already in use but are not yet popular enough to be described in great detail in an introductory textbook on parallel computing. Some of the approaches described have been developed in the area of high-performance computing, but they can also be used for small parallel systems, including multicore systems.

3.8.1.1 Unified Parallel C

Unified Parallel C (UPC) has been proposed as an extension to C for the use of parallel machines and cluster systems [47]. UPC is based on the model of a *partitioned global address space* (PGAS) [32], in which shared variables can be stored. Each such variable is associated with a certain thread, but the variable can also be read or manipulated by other threads. But typically, the access time for the variable is smaller for the associated thread than for another thread. Additionally, each thread can define private data to which it has exclusive access.

In UPC programs, parallel execution is obtained by creating a number of threads at program start. The UPC language extensions to C define a parallel execution model, memory consistency models for accessing shared variables, synchronization operations, and parallel loops. A detailed description is given in [47]. UPC compilers are available for several platforms. For Linux systems, free UPC compilers are

the Berkeley UPC compiler (see `upc.nersc.gov`) and the GCC UPC compiler (see `www.intrepid.com/upc3`). Other languages based on the PGAS model are the Co-Array Fortran Language (CAF), which is based on Fortran, and Titanium, which is similar to UPC, but is based on Java instead of C.

3.8.1.2 DARPA HPCS Programming Languages

In the context of the DARPA HPCS (*High Productivity Computing Systems*) program, new programming languages have been proposed and implemented, which support programming with a shared address space. These languages include Fortress, X10, and Chapel.

Fortress has been developed by Sun Microsystems. Fortress is a new object-oriented language based on Fortran which facilitates program development for parallel systems by providing a mathematical notation [11]. The language Fortress supports the parallel execution of programs by parallel loops and by the parallel evaluation of function arguments with multiple threads. Many constructs provided are implicitly parallel, meaning that the threads needed are created without an explicit control in the program.

A separate thread is, for example, implicitly created for each argument of a function call without any explicit thread creation in the program. Additionally, explicit threads can be created for the execution of program parts. Thread synchronization is performed with `atomic` expressions which guarantee that the effect on the memory becomes atomically visible immediately after the expression has been completely evaluated; see also the next section on transactional memory.

X10 has been developed by IBM as an extension to Java targeting at high-performance computing. Similar to UPC, X10 is based on the PGAS memory model and extends this model to the GALS model (*globally asynchronous, locally synchronous*) by introducing logical *places* [28]. The threads of a place have a locally synchronous view of their shared address space, but threads of different places work asynchronously with each other. X10 provides a variety of operations to access array variables and parts of array variables. Using array distributions, a partitioning of an array to different places can be specified. For the synchronization of threads, `atomic` blocks are provided which support an atomic execution of statements. By using `atomic` blocks, the details of synchronization are performed by the runtime system, and no low-level lock synchronization must be performed.

Chapel has been developed by Cray Inc. as a new parallel language for high-performance computing [37]. Some of the language constructs provided are similar to High-Performance Fortran (HPF). Like Fortress and X10, Chapel also uses the model of a global address space in which data structures can be stored and accessed. The parallel execution model supported is based on threads. At program start, there is a single main thread; using language constructs like parallel loops, more threads can be created. The threads are managed by the runtime system and the programmer does not need to start or terminate threads explicitly. For the synchronization of computations on shared data, synchronization variables and **atomic** blocks are provided.

3.8.1.3 Global Arrays

The global array (GA) approach has been developed to support program design for applications from scientific computing which mainly use array-based data structures, like vectors or matrices [127].

The GA approach is provided as a library with interfaces for C, C++, and Fortran for different parallel platforms. The GA approach is based on a global address space in which global array can be stored such that each process is associated with a logical block of the global array; access to this block is faster than access to the other blocks. The GA library provides basic operations (like put, get, scatter, gather) for the shared address space, as well as atomic operations and lock mechanisms for accessing global arrays. Data exchange between processes can be performed via global arrays. But a message-passing library like MPI can also be used. An important application area for the GA approach is the area of chemical simulations.

3.8.2 Transactional Memory

Threads must be synchronized when they access shared data concurrently. Standard approaches to avoid race conditions are **mutex variables** or **critical sections**. A typical programming style is as follows:

- The programmer identifies critical sections in the program and protects them with a mutex variable which is locked when the critical section is entered and unlocked when the critical section is left.
- This lock mechanism guarantees that the critical section is entered by one thread at a time, leading to *mutual exclusion*.

Using this approach with a lock mechanism leads to a sequentialization of the execution of critical sections. This may lead to performance problems and the critical sections may become a bottleneck. In particular, scalability problems often arise when a large number of threads are used and when the critical sections are quite large so that their execution takes quite long.

For small parallel systems like typical multicore architecture with only a few cores, this problem does not play an important role, since only a few threads are involved. But for large parallel systems of future multicore systems with a significantly larger number of cores, this problem must be carefully considered and the granularity of the critical section must be reduced significantly. Moreover, using a lock mechanism the programmer must strictly follow the conventions and must explicitly protect all program points at which an access conflict to shared data may occur in order to guarantee a correct behavior. If the programmer misses a program point which should be locked, the resulting program may cause error situations from time to time which depend on the relative execution speed of the threads and which are often not reproducible.

As an alternative approach to lock mechanisms, the use of **transactional memory** has been proposed, see, for example, [2, 16, 85]. In this approach, a program

is a series of transactions which appear to be executed indivisibly. A **transaction** is defined as a sequence of instructions which are executed by a single thread such that the following properties are fulfilled:

- **Serializability**: The transactions of a program appear to all threads to be executed in a global serial order. In particular, no thread observes an interleaving of the instructions of different transactions. All threads observe the execution of the transactions in the same global order.
- **Atomicity**: The updates in the global memory caused by the execution of the instructions of a transaction become atomically visible to the other threads after the executing thread has completed the execution of the instructions. A transaction that is completed successfully *commits*. If a transaction is interrupted, it has no effect on the global memory. A transaction that fails *aborts*. If a transaction fails, it is aborted for all threads, i.e., no thread observes any effect caused by the execution of the transaction. If a transaction is successful, it commits for all threads atomically.

Using a lock mechanism to protect a critical section does not provide atomicity in the sense just defined, since the effect on the shared memory becomes immediately visible. Using the concept of transactions for parallel programming requires the provision of new constructs which could, for example, be embedded into a programming language. A suitable construct is the use of `atomic` blocks where each `atomic` block defines a transaction [2]. The DARPA HPCS languages Fortran, X10, and Chapel contain such constructs to support the use of transactions, see Sect. 3.8.1.

The difference between the use of a lock mechanism and `atomic` blocks is illustrated in Fig. 3.19 for the example of a thread-safe access to a bank account using Java [2]. Access synchronization based on a lock mechanism is provided by the class `LockAccount`, which uses a `synchronized` block for accessing the account. When the method `add()` is called, this call is simply forwarded to the non-thread-safe `add()` method of the class `Account`, which we assume to be given. Executing the **synchronized** block causes an activation of the lock mechanism using the implicit mutex variable of the object `mutex`. This ensures the sequentialization of the access. An access based on transactions is implemented in the class `AtomicAccount`, which uses an `atomic` block to activate the non-thread-safe `add()` method of the `Account` class. The use of the `atomic` block ensures that the call to `add()` is performed atomically. Thus, the responsibility for guaranteeing serializability and atomicity is transferred to the runtime system. But depending on the specific situation, the runtime system does not necessarily need to enforce a sequentialization if this is not required. It should be noted that `atomic` blocks are not (yet) part of the Java language.

An important advantage of using transactions is that the runtime system can perform several transactions in parallel if the memory access pattern of the transactions allows this. This is not possible when using standard mutex variables. On the other hand, mutex variables can be used to implement more complex synchronization mechanisms which allow, e.g., a concurrent read access to shared data structures. An

```
class LockAccount implements Account {
  Object mutex;
  Account a;
  LockAccount (Account a) {
    this.a = a;
    mutex = New Object();
  }
  public int add (int x) {
    synchronized (mutex) {
       return a.add(x);
    }
  }
  ...
}

class AtomicAccount implements Account {
  Account a;
  AtomicAccount (Account a) {
    this.a = a;
  }
  public int add (int x) {
    atomic {
       return a.add(x);
    }
  }
  ...
}
```

Fig. 3.19 Comparison between a lock-oriented and a transaction-oriented implementation of an access to an account in Java

example is the read–write locks which allow multiple read accesses but only a single write access at a time, see Sect. 6.1.4 for an implementation in Pthreads. Since the runtime system can optimize the execution of transactions, using transactions may lead to a better scalability compared to the use of lock variables.

By using transactions, many responsibilities are transferred to the runtime system. In particular, the runtime system must ensure serializability and atomicity. To do so, the runtime system must provide the following two key mechanisms:

- **Version control**: The effect of a transaction must not become visible before the completion of the transaction. Therefore, the runtime system must perform the execution of the instructions of a transaction on a separate version of data. The previous version is kept as a copy in case the current transaction fails. If the current transaction is aborted, the previous version remains visible. If the current transaction commits, the new version becomes globally visible after the completion of the transaction.
- **Conflict detection**: To increase scalability, it is useful to execute multiple transactions in parallel. When doing so, it must be ensured that these transactions do not concurrently operate on the same data. To ensure the absence of such conflicts, the runtime system must inspect the memory access pattern of each transaction before issuing a parallel execution.

The use of transactions for parallel programming is an active area of research and the techniques developed are currently not available in standard programming languages. But transactional memory provides a promising approach, since it provides a more abstract mechanism than lock variables and can help to improve scalability of parallel programs for parallel systems with a shared address space like multicore processors. A detailed overview of many aspects of transactional memory can be found in [112, 144, 2].

3.9 Exercises for Chap. 3

Exercise 3.1 Consider the following sequence of instructions I_1, I_2, I_3, I_4, I_5:

I_1: $R_1 \leftarrow R_1 + R_2$
I_2: $R_3 \leftarrow R_1 + R_2$
I_3: $R_5 \leftarrow R_3 + R_4$
I_4: $R_4 \leftarrow R_3 + R_1$
I_5: $R_2 \leftarrow R_2 + R_4$

Determine all flow, anti, and output dependences and draw the resulting data dependence graph. Is it possible to execute some of these instructions parallel to each other?

Exercise 3.2 Consider the following two loops:

```
for (i=0 : n-1)      forall (i=0 : n-1)
  a(i) = b(i) +1;      a(i) = b(i) + 1;
  c(i) = a(i) +2;      c(i) = a(i) + 2;
  d(i) = c(i+1)+1;     d(i) = c(i+1) + 1;
endfor               endforall
```

Do these loops perform the same computations? Explain your answer.

Exercise 3.3 Consider the following sequential loop:

```
for (i=0 : n-1)
  a(i+1) = b(i) + c;
  d(i) = a(i) + e;
endfor
```

Can this loop be transformed into an equivalent `forall` loop? Explain your answer.

Exercise 3.4 Consider a 3×3 mesh network and the global communication operation scatter. Give a spanning tree which can be used to implement a scatter operation as defined in Sect. 3.5.2. Explain how the scatter operation is implemented on this tree. Also explain why the scatter operation is the dual operation of the gather operation and how the gather operation can be implemented.

Exercise 3.5 Consider a matrix of dimension 100×100. Specify the distribution vector $((p_1, b_1), (p_2, b_2))$ to describe the following data distributions for p processors:

- Column-cyclic distribution,
- Row-cyclic distribution,
- Blockwise column-cyclic distribution with block size 5,
- Blockwise row-cyclic distribution with block size 5.

Exercise 3.6 Consider a matrix of size 7×11. Describe the data distribution which results for the distribution vector ((2, 2), (3, 2)) by specifying which matrix element is stored by which of the six processors.

Exercise 3.7 Consider the matrix–vector multiplication programs in Sect. 3.6. Based on the notation used in this section, develop an SPMD program for computing a matrix–matrix multiplication $C = A \cdot B$ for a distributed address space. Use the notation from Sect. 3.6 for the communication operations. Assume the following distributions for the input matrices A and B:

(a) A is distributed in row-cyclic, B is distributed in column-cyclic order;
(b) A is distributed in column-blockwise, B in row-blockwise order;
(c) A and B are distributed in checkerboard order as has been defined on p. 114.

In which distribution is the result matrix C computed?

Exercise 3.8 The transposition of an $n \times n$ matrix A can be computed sequentially as follows:

```
for (i=0; i<n; i++)
   for (j=0; j<n; j++)
      B[i][j] = A[j][i];
```

where the result is stored in B. Develop an SPMD program for performing a matrix transposition for a distributed address space using the notation from Sect. 3.6. Consider both a row-blockwise and a checkerboard order distribution of A.

Exercise 3.9 The statement `fork(m)` creates `m` child threads $T_1, \ldots, T_m$ of the calling thread T, see Sect. 3.3.6, p. 109. Assume a semantics that a child thread executes the same program code as its parent thread starting at the program statement directly after the `fork()` statement and that a `join()` statement matches the last unmatched `fork()` statement. Consider a shared memory program fragment:

```
fork(3);
fork(2);
join();
join();
```

Give the tree of threads created by this program fragment.

Exercise 3.10 Two threads T_0 and T_1 access a shared variable in a critical section. Let `int flag[2]` be an array with `flag[i] = 1`, if thread `i` wants to enter the critical section. Consider the following approach for coordinating the access to the critical section:

Thread T_0

```
repeat {
   while (flag[1]) do no_op();
   flag[0] = 1;
   - - - critical section - - -;
   flag[0] = 0;
   - - - uncritical section - - -;
until 0;
```

Thread T_1

```
repeat {
   while (flag[0]) do no_op();
   flag[1] = 1;
   - - - critical section - - -;
   flag[1] = 0;
   - - - uncritical section - - -;
until 0;
```

Does this approach guarantee mutual exclusion, if both threads are executed on the same execution core? Explain your answer.

Exercise 3.11 Consider the following implementation of a lock mechanism:

```
int me;
int flag[2];
int lock() {
   int other = 1 - me;
   flag[me] = 1;
   while (flag[other]) ; // wait
}
int unlock() {
   flag[me] = 0;
}
```

Assume that two threads with ID 0 and 1 execute this piece of program to access a data structure concurrently and that each thread has stored its ID in its local variable `me`. Does this implementation guarantee mutual exclusion when the functions `lock()` and `unlock()` are used to protect critical sections? see Sect. 3.7.3. Can this implementation lead to a deadlock? Explain your answer.

Exercise 3.12 Consider the following example for the use of an atomic block [112]:

```
bool flag_A = false; bool flag_B = false;
```

Thread 1

```
atomic {
     while (!flag_A) ;
     flag_B = true;
}
```

Thread 2

```
atomic {
     flag_A = true ;
     while (!flag_B);
}
```

Why is this code incorrect?

Chapter 4
Performance Analysis of Parallel Programs

The most important motivation for using a parallel system is the reduction of the execution time of computation-intensive application programs. The execution time of a parallel program depends on many factors, including the architecture of the execution platform, the compiler and operating system used, the parallel programming environment and the parallel programming model on which the environment is based, as well as properties of the application program such as locality of memory references or dependencies between the computations to be performed. In principle, all these factors have to be taken into consideration when developing a parallel program. However, there may be complex interactions between these factors, and it is therefore difficult to consider them all.

To facilitate the development and analysis of parallel programs, *performance measures* are often used which abstract from some of the influencing factors. Such performance measures can be based not only on theoretical cost models but also on measured execution times for a specific parallel system.

In this chapter, we consider performance measures for an analysis and comparison of different versions of a parallel program in more detail. We start in Sect. 4.1 with a discussion of different methods for a performance analysis of (sequential and parallel) execution platforms, which are mainly directed toward a performance evaluation of the architecture of the execution platform, without considering a specific user-written application program. In Sect. 4.2, we give an overview of popular performance measures for parallel programs, such as speedup or efficiency. These performance measures mainly aim at a comparison of the execution time of a parallel program with the execution time of a corresponding sequential program. Section 4.3 analyzes the running time of global communication operations, such as broadcast or scatter operations, in the distributed memory model with different interconnection networks. Optimal algorithms and asymptotic running times are derived. In Sect. 4.4, we show how runtime functions (in closed form) can be used for a runtime analysis of application programs. This is demonstrated for parallel computations of a scalar product and of a matrix–vector multiplication. Section 4.5 contains a short overview of popular theoretical cost models like BSP and LogP.

T. Rauber, G. Rünger, *Parallel Programming*,
DOI 10.1007/978-3-642-04818-0_4, © Springer-Verlag Berlin Heidelberg 2010

4.1 Performance Evaluation of Computer Systems

The performance of a computer system is one of the most important aspects of its evaluation. Depending on the point of view, different criteria are important to evaluate performance. The user of a computer system is interested in small **response times**, where the response time of a program is defined as the time between the start and the termination of the program. On the other hand, a large computing center is mainly interested in high **throughputs**, where the throughput is the average number of work units that can be executed per time unit.

4.1.1 Evaluation of CPU Performance

In the following, we first consider a *sequential* computer system and use the response times as performance criteria. The performance of a computer system becomes larger, if the response times for a given set of application programs become smaller. The response time of a program A can be split into

- the **user CPU time** of A, capturing the time that the CPU spends for executing A;
- the **system CPU time** of A, capturing the time that the CPU spends for the execution of routines of the operating system issued by A;
- the **waiting time** of A, caused by waiting for the completion of I/O operations and by the execution of other programs because of time sharing.

So the response time of a program includes the waiting times, but these waiting times are not included in the CPU time. For Unix systems, the `time` command can be used to get information on the fraction of the CPU and waiting times of the overall response time. In the following, we ignore the waiting times, since these strongly depend on the load of the computer system. We also neglect the system CPU time, since this time mainly depends on the implementation of the operating system, and concentrate on the execution times that are directly caused by instructions of the application program [137].

The user CPU time depends both on the translation of the statements of the program into equivalent sequences of instructions by the compiler and on the execution time for the single instructions. The latter time is strongly influenced by the **cycle time** of the CPU (also called *clock cycle time*), which is the reciprocal of the **clock rate**. For example, a processor with a clock rate of 2 GHz $= 2 \cdot 10^9 \cdot 1/\text{s}$ has cycle time of $1/(2 \cdot 10^9)\text{s} = 0.5 \cdot 10^{-9}\,\text{s} = 0.5\,\text{ns}$ (s denotes seconds and ns denotes nanoseconds). In the following, the cycle time is denoted as t_{cycle} and the user CPU time of a program A is denoted as $T_{\text{U\_CPU}}(A)$. This time is given by the product of t_{cycle} and the total number $n_{\text{cycle}}(A)$ of CPU cycles needed for all instructions of A:

$$T_{\text{U\_CPU}}(A) = n_{\text{cycle}}(A) \cdot t_{\text{cycle}} \; . \tag{4.1}$$

Different instructions may have different execution times. To get a relation between the number of cycles and the number of instructions executed for program A, the

average number of CPU cycles used for instructions of program A is considered. This number is called **CPI** (**C**lock cycles **P**er **I**nstruction). The CPI value depends on the program A to be executed, since the specific selection of instructions has an influence on CPI. Thus, for the same computer system, different programs may lead to different CPI values. Using CPI, the user CPU time of a program A can be expressed as

$$T_{\text{U\_CPU}}(A) = n_{\text{instr}}(A) \cdot CPI(A) \cdot t_{\text{cycle}}, \tag{4.2}$$

where $n_{\text{instr}}(A)$ denotes the total number of instructions executed for A. This number depends on many factors. The architecture of the computer system has a large influence on $n_{\text{instr}}(A)$, since the behavior of the instruction provided by the architecture determines how efficient constructs of the programming language can be translated into sequences of instructions. Another important influence comes from the *compiler*, since the compiler selects the instructions to be used in the machine program. An efficient compiler can make the selection such that a small number $n_{\text{instr}}(A)$ results.

For a given program, the CPI value strongly depends on the implementation of the instructions, which depends on the internal organization of the CPU and the memory system. The CPI value also depends on the compiler, since different instructions may have different execution times and since the compiler can select instructions such that a smaller or a larger CPI value results.

We consider a processor which provides n types of instructions, $I_1, \ldots, I_n$. The average number of CPU cycles needed for instructions of type I_i is denoted by CPI_i, and $n_i(A)$ is the number of instructions of type I_i executed for a program A, $i = 1, \ldots, n$. Then the total number of CPU cycles used for the execution of A can be expressed as

$$n_{\text{cycle}}(A) = \sum_{i=1}^{n} n_i(A) \cdot CPI_i. \tag{4.3}$$

The total number of machine instructions executed for a program A is an exact measure of the number of CPU cycles and the resulting execution time of A only if all instructions require the same number of CPU cycles, i.e., have the same values for CPI_i. This is illustrated by the following example, see [137].

Example We consider a processor with three instruction classes $\mathcal{I}_1, \mathcal{I}_2, \mathcal{I}_3$ containing instructions which require 1, 2, or 3 cycles for their execution, respectively. We assume that there are two different possibilities for the translation of a programming language construct using different instructions according to the following table:

	Instruction classes				
Translation	$\mathcal{I}_1$	$\mathcal{I}_2$	$\mathcal{I}_3$	Sum of the instructions	n_{cycle}
1	2	1	2	5	10
2	4	1	1	6	9

Translation 2 needs less cycles than translation 1, although translation 2 uses a larger number of instructions. Thus, translation 1 leads to a CPI value of $10/5 = 2$, whereas translation 2 leads to a CPI value of $9/6 = 1.5$.

4.1.2 MIPS and MFLOPS

A performance measure that is sometimes used in practice to evaluate the performance of a computer system is the **MIPS rate** (**M**illion **I**nstructions **P**er **S**econd). Using the notation from the previous subsection for the number of instructions $n_{\text{instr}}(A)$ of a program A and for the user CPU time $T_{\text{U\_CPU}}(A)$ of A, the MIPS rate of A is defined as

$$MIPS(A) = \frac{n_{\text{instr}}(A)}{T_{\text{U\_CPU}}(A) \cdot 10^6} \quad . \tag{4.4}$$

Using Eq. (4.2), this can be transformed into

$$MIPS(A) = \frac{r_{\text{cycle}}}{CPI(A) \cdot 10^6} \ ,$$

where $r_{\text{cycle}} = 1/t_{\text{cycle}}$ is the clock rate of the processor. Therefore, faster processors lead to larger MIPS rates than slower processors. Because the CPI value depends on the program A to be executed, the resulting MIPS rate also depends on A.

Using MIPS rates as performance measure has some drawbacks. First, the MIPS rate only considers the *number* of instructions. But more powerful instructions usually have a longer execution time, but fewer of such powerful instructions are needed for a program. This favors processors with simple instructions over processors with more complex instructions. Second, the MIPS rate of a program does not necessarily correspond to its execution time: Comparing two programs A and B on a processor X, it can happen that B has a higher MIPS rate than A, but A has a smaller execution time. This can be illustrated by the following example.

Example Again, we consider a processor X with three instruction classes $\mathcal{I}_1, \mathcal{I}_2, \mathcal{I}_3$ containing instructions which require 1, 2, or 3 cycles for their execution, respectively.

We assume that processor X has a clock rate of 2 GHz and, thus, the cycle time is 0.5 ns. Using two different compilers for the translation of a program may lead to two different machine programs A_1 and A_2 for which we assume the following numbers of instructions from the different classes:

Program	$\mathcal{I}_1$	$\mathcal{I}_2$	$\mathcal{I}_3$
A_1	$5 \cdot 10^9$	$1 \cdot 10^9$	$1 \cdot 10^9$
A_2	$10 \cdot 10^9$	$1 \cdot 10^9$	$1 \cdot 10^9$

For the CPU time of A_j, $j = 1, 2$, we get from Eqs. (4.2) and (4.3)

$$T_{\text{U\_CPU}}(A_j) = \sum_{i=1}^{3} n_i(A_j) \cdot CPI_i(A_j) \cdot t_{\text{cycle}},$$

where $n_i(A_j)$ is the number of instruction executions from the table and $CPI_i(A_j)$ is the number of cycles needed for instructions of class $\mathcal{I}_i$ for $i = 1, 2, 3$. Thus, machine program A_1 leads to an execution time of 5 s, whereas A_2 leads to an execution time of 7.5 s. The MIPS rates of A_1 and A_2 can be computed with Eq. (4.4). For A_1, in total $7 \cdot 10^9$ instructions are executed, leading to a MIPS rate of 1400 (1/s). For A_2, a MIPS rate of 1600 (1/s) results. This shows that A_2 has a higher MIPS rate than A_1, but A_1 has a smaller execution time. □

For program with scientific computations, the MFLOPS rate (**M**illion **F**loating-point **O**perations **P**er **S**econd) is sometimes used. The MFLOPS rate of a program A is defined by

$$MFLOPS(A) = \frac{n_{\text{flp\_op}}(A)}{T_{\text{U\_CPU}}(A) \cdot 10^6} \text{ [1/s]}, \tag{4.5}$$

where $n_{\text{flp\_op}}(A)$ is the number of floating-point operations executed by A. The MFLOPS rate is not based on the number of instructions executed, as is the case for the MIPS rate, but on the number of arithmetic operations on floating-point values performed by the execution of their instructions. Instructions that do not perform floating-point operations have no effect on the MFLOPS rate. Since the effective number of operations performed is used, the MFLOPS rate provides a fair comparison of different program versions performing the same operations, and larger MFLOPS rates correspond to faster execution times.

A drawback of using the MFLOPS rate as performance measure is that there is no differentiation between different types of floating-point operations performed. In particular, operations like division and square root that typically take quite long to perform are counted in the same way as operations like addition and multiplication that can be performed much faster. Thus, programs with simpler floating-point operations are favored over programs with more complex operations. However, the MFLOPS rate is well suited to compare program versions that perform the same floating-point operations.

4.1.3 Performance of Processors with a Memory Hierarchy

According to Eq. (4.1), the user CPU time of a program A can be represented as the product of the number of CPU cycles $n_{\text{cycles}}(A)$ for A and the cycle time t_{cycle} of the processor. By taking the access time to the memory system into consideration, this can be refined to

$$T_{\text{U\_CPU}}(A) = (n_{\text{cycles}}(A) + n_{\text{mm\_cycles}}(A)) \cdot t_{\text{cycle}}, \tag{4.6}$$

where $n_{\text{mm\_cycles}}(A)$ is the number of additional machine cycles caused by memory accesses of A. In particular, this includes those memory accesses that lead to the loading of a new cache line because of a cache miss, see Sect. 2.7. We first consider a one-level cache. If we assume that cache hits do not cause additional machine cycles, they are captured by $n_{\text{cycles}}(A)$. Cache misses can be caused by read misses or write misses:

$$n_{\text{mm\_cycles}}(A) = n_{\text{read\_cycles}}(A) + n_{\text{write\_cycles}}(A).$$

The number of cycles needed for read accesses can be expressed as

$$n_{\text{read\_cycles}}(A) = n_{\text{read\_op}}(A) \cdot r_{\text{read\_miss}}(A) \cdot n_{\text{miss\_cycle}},$$

where $n_{\text{read\_op}}(A)$ is the total number of read operations of A, $r_{\text{read\_miss}}(A)$ is the read miss rate for A, and $n_{\text{miss\_cycle}}$ is the number of machine cycles needed to load a cache line into the cache in case of a read miss; this number is also called *read miss penalty*. A similar expression can be given for the number of cycles $n_{\text{write\_cycles}}$ needed for write accesses. The effect of read and write misses can be combined for simplicity which results in the following expression for the user CPU time:

$$T_{\text{U\_CPU}}(A) = n_{\text{instr}}(A) \cdot (CPI(A) + n_{\text{rw\_op}}(A) \cdot r_{\text{miss}}(A) \cdot n_{\text{miss\_cycle}}) \cdot t_{\text{cycle}}, \tag{4.7}$$

where $n_{\text{rw\_op}}(A)$ is the total number of read or write operations of A, $r_{\text{miss}}(A)$ is the (read and write) miss rate of A, and $n_{\text{miss\_cycles}}$ is the number of additional cycles needed for loading a new cache line. Equation (4.7) is derived from Eqs. (4.2) and (4.6).

Example We consider a processor for which each instruction takes two cycles to execute, i.e., it is CPI $= 2$, see [137]. The processor uses a cache for which the loading of a cache block takes 100 cycles. We consider a program A for which the (read and write) miss rate is 2% and in which 33% of the instructions executed are load and store operations, i.e., it is $n_{\text{rw\_op}}(A) = n_{\text{instr}}(A) \cdot 0.33$. According to Eq. (4.7) it is

$$\begin{aligned} T_{\text{U\_CPU}}(A) &= n_{\text{instr}}(A) \cdot (2 + 0.33 \cdot 0.02 \cdot 100) \cdot t_{\text{cycle}} \\ &= n_{\text{instr}}(A) \cdot 2.66 \cdot t_{\text{cycle}}. \end{aligned}$$

This can be interpreted such that the ideal CPI value of 2 is increased to the real CPI value of 2.66 if the data cache misses are taken into consideration. This does not take instruction cache misses into consideration. The equation for $T_{\text{U\_CPU}}(A)$ can also be used to compute the benefit of using a data cache: Without a data cache, each memory access would take 100 cycles, leading to a real CPI value of $2 + 100 \cdot 0.33 = 35$.

Doubling the clock rate of the processor without changing the memory system leads to an increase in the cache loading time to 200 cycles, resulting in a real CPI value of $2 + 0.33 \cdot 0.02 \cdot 200 = 3.32$. Using t_{cycle} for the original cycle time, the CPU time on the new processor with half of the cycle time yields

$$\tilde{T}_{\text{U\_CPU}}(A) = n_{\text{instr}}(A) \cdot 3.32 \cdot t_{\text{cycle}}/2.$$

Thus, the new processor needs 1.66 instead of 2.66 original cycle time units. Therefore, doubling the clock rate of the processor leads to a decrease of the execution time of the program to 1.66/2.66, which is about 62.4% of the original execution time, but not 50% as one might expect. This shows that the memory system has an important influence on program execution time. □

The influence of memory access times using a memory hierarchy can be captured by defining an *average memory access time* [137]. The average read access time $t_{\text{read\_access}}(A)$ of a program A can be defined as

$$t_{\text{read\_access}}(A) = t_{\text{read\_hit}} + r_{\text{read\_miss}}(A) \cdot t_{\text{read\_miss}}, \tag{4.8}$$

where $t_{\text{read\_hit}}$ is the time for a read access to the cache. The additional time needed for memory access in the presence of cache misses can be captured by multiplying the cache read miss rate $r_{\text{read\_miss}}(A)$ with the read miss penalty time $t_{\text{read\_miss}}$ needed for loading a cache line. In Eq. (4.7), $t_{\text{read\_miss}}$ has been calculated from $n_{\text{miss\_cycle}}$ and t_{cycle}. The time $t_{\text{read\_hit}}$ for a read hit in the cache was assumed to be included in the time for the execution of an instruction.

It is beneficial if the access time to the cache is adapted to the cycle time of the processor, since this avoids delays for memory accesses in case of cache hits. To do this, the first-level (L1) cache must be kept small and simple and an additional second-level (L2) cache is used, which is large enough such that most memory accesses go to the L2 cache and not to main memory. For performance analysis, the modeling of the average read access time is based on the performance values of the L1 cache. In particular, for Eq. (4.8), we have

$$t_{\text{read\_access}}(A) = t^{(L1)}_{\text{read\_hit}} + r^{(L1)}_{\text{read\_miss}}(A) \cdot t^{(L1)}_{\text{read\_miss}},$$

where $r^{(L1)}_{\text{read\_miss}}(A)$ is the cache read miss rate of A for the L1 cache, calculated by dividing the total number of read accesses causing an L1 cache miss by the total number of read accesses. To model the reload time $t^{(L1)}_{\text{read\_miss}}$ of the L1 cache, the access time and miss rate of the L2 cache can be used. More precisely, we get

$$t^{L1}_{\text{read\_miss}} = t^{(L2)}_{\text{read\_hit}} + r^{(L2)}_{\text{read\_miss}}(A) \cdot t^{(L2)}_{\text{read\_miss}},$$

where $r^{(L2)}_{\text{read\_miss}}(A)$ is the read miss rate of A for the L2 cache, calculated by dividing the total number of read misses of the L2 cache by the total number of read misses

of the L1 cache. Thus, the global read miss rate of program A can be calculated by $r^{(L1)}_{\text{read\_miss}}(A) \cdot r^{(L2)}_{\text{read\_miss}}(A)$.

4.1.4 Benchmark Programs

The performance of a computer system may vary significantly, depending on the program considered. For two programs A and B, the following situation can occur: Program A has a smaller execution time on a computer system X than on a computer system Y, whereas program B has a smaller execution time on Y than on X.

For the user, it is important to base the selection of a computer system on a set of programs that are often executed by the user. These programs may be different for different users. Ideally, the programs would be weighted by their execution time and their execution frequency. But often, the programs to be executed on a computer system are not known in advance. Therefore, **benchmark programs** have been developed which allow a standardized performance evaluation of computer systems based on specific characteristics that can be measured on a given computer system. Different benchmark programs have been proposed and used, including the following approaches, listed in increasing order of their usefulness:

- **Synthetic benchmarks**, which are typically small artificial programs containing a mixture of statements which are selected such that they are representative for a large class of real applications. Synthetic benchmarks usually do not execute meaningful operations on a large set of data. This bears the risk that some program parts may be removed by an optimizing compiler. Examples for synthetic benchmarks are *Whetstone* [36, 39], which has originally been formulated in Fortran to measure floating-point performance, and *Dhrystone* [174] to measure integer performance in C. The performance measured by Whetstone or Dhrystone is measured in specific units as `KWhetstone/s` or `KDhrystone/s`. The largest drawback of synthetic benchmarks is that they are not able to match the profile and behavior of large application programs with their complex interactions between computations of the processor and accesses to the memory system. Such interactions have a large influence on the resulting performance, yet they cannot be captured by synthetic benchmarks. Another drawback is that a compiler or system can be tuned toward simple benchmark programs to let the computer system appear faster than it is for real applications.
- **Kernel benchmarks** with small but relevant parts of real applications which typically capture a large portion of the execution time of real applications. Compared to real programs, kernels have the advantage that they are much shorter and easier to analyze. Examples for kernel collections are the *Livermore Loops* (Livermore Fortran Kernels, LFK) [121, 50], consisting of 24 loops extracted from scientific simulations, and *Linpack* [41] capturing a piece of a Fortran library with linear algebra computations. Both kernels compute the performance in MFLOPS. The drawback of kernels is that the performance values they produce are often too large for applications that come from other areas than scientific computing.

A variant of kernels is a collection of toy programs, which are small, but complete programs performing useful computations. Examples are quicksort for sorting or the sieve of Erathostenes for prime test.

- **Real application benchmarks** comprise several entire programs which reflect a workload of a standard user. Such collections are often called *benchmark suites*. They have the advantage that all aspects of the selected programs are captured. The performance results produced are meaningful for users for which the benchmark suite is representative for typical workloads. Examples for benchmark suites are the SPEC benchmarks, described in the following, for desktop computers, and the EEMBC benchmarks (EDV Embedded Microprocessor Benchmark Consortium) for embedded systems, see `www.eembc.org` for more information.

The most popular benchmark suite is the SPEC benchmark suite (System Performance Evaluation Cooperation), see `www.spec.org` for detailed information. The cooperation was founded in 1988 with the goal to define a standardized performance evaluation method for computer systems and to facilitate a performance comparison. Until now, SPEC has published five generations of benchmark suites for desktop computers: SPEC89, SPEC92, SPEC95, SPEC00, and SPEC06. There are other benchmark suites for file servers (SPECSFC), web servers (SPECWeb), or parallel systems like SPECOpenMP.

SPEC06 is the current version for desktop computers. It consists of 12 integer programs (9 written in C, 3 in C++) and 17 floating-point programs (6 written in Fortran, 3 in C, 4 in C++, and 4 in mixed C and Fortran). The integer programs include, for example, a compression program (bzip2), a C compiler (gcc), a video compression program, a chess game, and an XML parser. The floating-point programs include, for example, several simulation programs from physics, a speech recognition program, a ray-tracing program (povray), as well as programs from numerical analysis and a linear programming algorithm (soplex).

The SPEC integer and floating-point programs are used to compute two performance measures SPECint2006 and SPECfp2006, respectively, to express the average integer and floating-point performance of a specific computer system. The performance measures are given as the relative performance with respect to a fixed reference computer, specified by the SPEC suite. For SPEC06, the reference computer is a Sun Ultra Enterprise 2 with a 296 MHz UltraSparc II processor. This reference computer gets a SPECint2006 and SPECfp2006 score of 1.0. Larger values of the performance measures correspond to a higher performance of the computer system tested. The SPECint2006 and SPECfp2006 values are determined separately by using the SPEC integer and floating-point programs, respectively. To perform the benchmark evaluation and to compute the performance measures SPECint2006 or SPECfp2006, the following three steps are executed:

(1) Each of the programs is executed three times on the computer system U to be tested. For each of the programs A_i an average execution time $T_U(A_i)$ in seconds is determined by taking the median of the three execution times measured, i.e., the middle value.

(2) For each program, the execution time $T_U(A_i)$ determined in step (1) is normalized with respect to the reference computer R by dividing the execution time $T_R(A_i)$ on R by the execution time $T_U(A_i)$ on U. This yields an execution factor $F_U(A_i) = T_R(A_i)/T_U(A_i)$ for each of the programs A_i which expresses how much faster machine U is compared to R for program A_i.
(3) SPECint2006 is computed as the geometric mean of the execution factors of the 12 SPEC integer programs, i.e., a global factor G_U^{int} is computed by

$$G_U^{int} = \sqrt[12]{\prod_{i=1}^{12} F_U(A_i)}.$$

G_U^{int} is the SPECint2006 score, expressing how much faster U is compared to R. SPECfp2006 is defined similarly, using the geometric mean of the 17 floating-point programs.

An alternative to the geometric means would be the *arithmetic means* to compute the global execution factors, by calculation, for example, $A_U^{int} = 1/12 \sum_{i=1}^{12} F_U(A_i)$. But using the geometric means has some advantages. The most important advantage is that the comparison between two machines is independent of the choice of the reference computer. This is not necessarily the case when the arithmetic means is used instead; this is illustrated by a following example calculation.

Example Two programs A_1 and A_2 and two machines X and Y are considered, see also [84]. Assuming the following execution times

$$\begin{array}{ll} T_X(A_1) = 1\,\text{s}, & T_Y(A_1) = 10\,\text{s}, \\ T_X(A_2) = 500\,\text{s}, & T_Y(A_2) = 50\,\text{s} \end{array}$$

results in the following execution factors if Y is used as reference computer:

$$F_X(A_1) = 10,\ F_X(A_2) = 0.1,\ F_Y(A_1) = F_Y(A_2) = 1\ .$$

This yields the following performance score for the arithmetic means A and the geometric means G:

$$G_X = \sqrt{10 \cdot 0.1} = 1,\ A_X = \frac{1}{2}(10 + 0.1) = 5.05,\ G_Y = A_Y = 1\ .$$

Using X as reference computer yields the following execution factors:

$$F_X(P_1) = F_X(P_2) = 1,\ F_Y(P_1) = 0.1,\ F_Y(P_2) = 10$$

resulting in the following performance scores:

$$G_X = 1, A_X = 1, G_Y = \sqrt{10 \cdot 0.1} = 1, A_Y = \frac{1}{2}(0.1 + 10) = 5.05.$$

Thus, considering the arithmetic means, using Y as reference computer yields the statement that $X = 5.05$ times faster than Y. Using X as reference computer yields the opposite result. Such contradictory statements are avoided by using the geometric means, which states that X and Y have the same performance, independently of the reference computer.

A drawback of the geometric means is that it does not provide information about the actual execution time of the programs. This can be seen from the example just given. Executing A_1 and A_2 only once requires 501 s on X and 60 s on Y, i.e., Y is more than eight times faster than X. □

A detailed discussion of benchmark programs and program optimization issues can be found in [42, 92, 69], which also contain references to other literature.

4.2 Performance Metrics for Parallel Programs

An important criterion for the usefulness of a parallel program is its runtime on a specific execution platform. The **parallel runtime** $T_p(n)$ of a program is the time between the start of the program and the end of the execution on all participating processors; this is the point in time when the last processor finishes its execution for this program. The parallel runtime is usually expressed for a specific number p of participating processors as a function of the problem size n. The problem size is given by the size of the input data, which can for example be the number of equations of an equation system to be solved. Depending on the architecture of the execution platform, the parallel runtime comprises the following times:

- the runtime for the execution of *local computations* of each participating processor; these are the computations that each processor performs using data in its local memory;
- the runtime for the *exchange of data* between processors, e.g., by performing explicit communication operations in the case of a distributed address space;
- the runtime for the *synchronization* of the participating processors when accessing shared data structures in the case of a shared address space;
- *waiting times* occurring because of an unequal load distribution of the processors; waiting times can also occur when a processor has to wait before it can access a shared data structure to ensure mutual exclusion.

The time spent for data exchange and synchronization as well as waiting times can be considered as overhead since they do not contribute directly to the computations to be performed.

4.2.1 Speedup and Efficiency

The cost of a parallel program captures the runtime that each participating processor spends for executing the program.

4.2.1.1 Cost of a Parallel Program

The cost $C_p(n)$ of a parallel program with input size n executed on p processors is defined by

$$C_p(n) = p \cdot T_p(n).$$

Thus, $C_p(n)$ is a measure of the total amount of work performed by all processors. Therefore, the cost of a parallel program is also called *work* or processor–runtime product.

A parallel program is called **cost-optimal** if $C_p(n) = T^\star(n)$, i.e., if it executes the same total number of operations as the fastest sequential program which has runtime $T^\star(n)$. Using asymptotic execution times, this means that a parallel program is cost-optimal if $T^\star(n)/C_p(n) \in \Theta(1)$ (see Sect. 4.3.1 for the Θ definition).

4.2.1.2 Speedup

For the analysis of parallel programs, a comparison with the execution time of a sequential implementation is especially important to see the benefit of parallelism. Such a comparison is often based on the relative saving in execution time as expressed by the notion of speedup. The speedup $S_p(n)$ of a parallel program with parallel execution time $T_p(n)$ is defined as

$$S_p(n) = \frac{T^*(n)}{T_p(n)},$$

where p is the number of processors used to solve a problem of size n. $T^\star(n)$ is the execution time of the best sequential implementation to solve the same problem. The speedup of a parallel implementation expresses the relative saving of execution time that can be obtained by using a parallel execution on p processors compared to the best sequential implementation. The concept of speedup is used both for a theoretical analysis of algorithms based on the asymptotic notation and for the practical evaluation of parallel programs.

Theoretically, $S_p(n) \leq p$ always holds, since for $S_p(n) > p$, a new sequential algorithm could be constructed which is faster than the sequential algorithm that has been used for the computation of the speedup. The new sequential algorithm is derived from the parallel algorithm by a round robin simulation of the steps of the participating p processors, i.e., the new sequential algorithm uses its first p steps to simulate the first step of all p processors in a fixed order. Similarly, the next p steps are used to simulate the second step of all p processors, and so on. Thus, the

new sequential algorithm performs p times more steps than the parallel algorithm. Because of $S_p(n) > p$, the new sequential algorithm would have execution time

$$p \cdot T_p(n) = p \cdot \frac{T^*(n)}{S_p(n)} < T^*(n).$$

This is a contradiction to the assumption that the best sequential algorithm has been used for the speedup computation. The new algorithm is faster.

The speedup definition given above requires a comparison with the fastest sequential algorithm. This algorithm may be difficult to determine or construct. Possible reasons may be as follows:

- The best sequential algorithm may not be known. There might be the situation that a lower bound for the execution time of a solution method for a given problem can be determined, but until now, no algorithm with this asymptotic execution time has yet been constructed.
- There exists an algorithm with the optimum asymptotic execution time, but depending on the size and the characteristics of a specific input set, other algorithms lead to lower execution times in practice. For example, the use of balanced trees for the dynamic management of data sets should be preferred only if the data set is large enough and if enough access operations are performed.
- The sequential algorithm which leads to the smallest execution times requires a large effort to be implemented.

Because of these reasons, the speedup is often computed by using a sequential version of the parallel implementation instead of the best sequential algorithm.

In practice, superlinear speedup can sometimes be observed, i.e., $S_p(n) > p$ can occur. The reason for this behavior often lies in cache effects: A typical parallel program assigns only a fraction of the entire data set to each processor. The fraction is selected such that the processor performs its computations on its assigned data set. In this situation, it can occur that the entire data set does not fit into the cache of a single processor executing the program sequentially, thus leading to cache misses during the computation. But when several processors execute the program with the same amount of data in parallel, it may well be that the fraction of the data set assigned to each processor fits into its local cache, thus avoiding cache misses. However, superlinear speedup does not occur often. A more typical situation is that a parallel implementation does not even reach *linear speedup* ($S_p(n) = p$), since the parallel implementation requires additional overhead for the management of parallelism. This overhead might be caused by the necessity to exchange data between processors, by synchronization between processors, or by waiting times caused by an unequal load balancing between the processors. Also, a parallel program might have to perform more computations than the sequential program version because replicated computations are performed to avoid data exchanges. The parallel program might also contain computations that must be executed sequentially by only one of the processors because of data dependencies. During such sequential

computations, the other processors must wait. Input and output operations are a typical example for sequential program parts.

4.2.1.3 Efficiency

An alternative measure for the performance of a parallel program is the efficiency. The efficiency captures the fraction of time for which a processor is usefully employed by computations that also have to be performed by a sequential program. The definition of the efficiency is based on the cost of a parallel program and can be expressed as

$$E_p(n) = \frac{T^*(n)}{C_p(n)} = \frac{S_p(n)}{p} = \frac{T^*(n)}{p \cdot T_p(n)},$$

where $T^\star(n)$ is the sequential execution time of the best sequential algorithm and $T_p(n)$ is the parallel execution time on p processors. If no superlinear speedup occurs, then $E_p(n) \leq 1$. An ideal speedup $S_p(n) = p$ corresponds to an efficiency of $E_p(n) = 1$.

4.2.1.4 Amdahl's Law

The parallel execution time of programs cannot be arbitrarily reduced by employing parallel resources. As shown, the number of processors is an upper bound for the speedup that can be obtained. Other restrictions may come from data dependencies within the algorithm to be implemented, which may limit the degree of parallelism. An important restriction comes from program parts that have to be executed sequentially. The effect on the obtainable speedup can be captured quantitatively by **Amdahl's law** [15]:

When a (constant) fraction f, $0 \leq f \leq 1$, of a parallel program must be executed sequentially, the parallel execution time of the program is composed of a fraction of the sequential execution time $f \cdot T^\star(n)$ and the execution time of the fraction $(1 - f) \cdot T^\star(n)$, fully parallelized for p processors, i.e., $(1 - f)/p \cdot T^\star(n)$. The attainable speedup is therefore

$$S_p(n) = \frac{T^*(n)}{f \cdot T^*(n) + \frac{1-f}{p} T^*(n)} = \frac{1}{f + \frac{1-f}{p}} \leq \frac{1}{f}.$$

This estimation assumes that the best sequential algorithm is used and that the parallel part of the program can be perfectly parallelized. The effect of the sequential computations on the attainable speedup can be demonstrated by considering an example: If 20% of a program must be executed sequentially, then the attainable speedup is limited to $1/f = 5$ according to Amdahl's law, no matter how many processors are used. Program parts that must be executed sequentially must be taken into account in particular when a large number of processors are employed.

4.2.2 Scalability of Parallel Programs

The scalability of a parallel program captures the performance behavior for an increasing number of processors.

4.2.2.1 Scalability

Scalability is a measure describing whether a performance improvement can be reached that is proportional to the number of processors employed. Scalability depends on several properties of an algorithm and its parallel execution. Often, for a fixed problem size n a saturation of the speedup can be observed when the number p of processors is increased. But increasing the problem size for a fixed number of processors usually leads to an increase in the attained speedup. In this sense, scalability captures the property of a parallel implementation that the efficiency can be kept constant if both the number p of processors and the problem size n are increased. Thus, scalability is an important property of parallel programs since it expresses that larger problems can be solved in the same time as smaller problems if a sufficiently large number of processors are employed.

The increase in the speedup for increasing problem size n cannot be captured by Amdahl's law. Instead, a variant of Amdahl's law can be used which assumes that the sequential program part is not a constant fraction f of the total amount of computations, but that it decreases with the input size. In this case, for an arbitrary number p of processors, the intended speedup $\leq p$ can be obtained by setting the problem size to a large enough value.

4.2.2.2 Gustafson's Law

This behavior is expressed by Gustafson's law [78] for the special case that the sequential program part has a *constant* execution time, independent of the problem size. If τ_f is the constant execution time of the sequential program part and $\tau_v(n, p)$ is the execution time of the parallelizable program part for problem size n and p processors, then the **scaled speedup** of the program is expressed by

$$S_p(n) = \frac{\tau_f + \tau_v(n, 1)}{\tau_f + \tau_v(n, p)}.$$

If we assume that the parallel program is perfectly parallelizable, then $\tau_v(n, 1) = T^*(1) - \tau_f$ and $\tau_v(n, p) = (T^*(n) - \tau_f)/p$ follow and thus

$$S_p(n) = \frac{\tau_f + T^*(n) - \tau_f}{\tau_f + (T^*(n) - \tau_f)/p} = \frac{\frac{\tau_f}{T^*(n)-\tau_f} + 1}{\frac{\tau_f}{T^*(n)-\tau_f} + \frac{1}{p}},$$

and therefore

$$\lim_{n\to\infty} S_p(n) = p,$$

if $T^\star(n)$ increases strongly monotonically with n. This is for example true for $\tau_v(n, p) = n^2/p$, which describes the amount of parallel computations for many iteration methods on two-dimensional meshes:

$$\lim_{n\to\infty} S_p(n) = \lim_{n\to\infty} \frac{\tau_f + n^2}{\tau_f + n^2/p} = \lim_{n\to\infty} \frac{\tau_f/n^2 + 1}{\tau_f/n^2 + 1/p} = p.$$

There exist more complex scalability analysis methods which try to capture how the problem size n must be increased relative to the number p of processors to obtain a constant efficiency. An example is the use of isoefficiency functions as introduced in [75] which express the required change of the problem size n as a function of the number of processors p.

4.3 Asymptotic Times for Global Communication

In this section, we consider the analytical modeling of the execution time of parallel programs. For the implementation of parallel programs, many design decisions have to be made concerning, for example, the distribution of program data and the mapping of computations to resources of the execution platform. Depending on these decisions, different communication or synchronization operations must be performed, and different load balancing may result, leading to different parallel execution times for different program versions. Analytical modeling can help to perform a pre-selection by determining which program versions are promising and which program versions lead to significantly larger execution times, e.g., because of a potentially large communication overhead. In many situations, analytical modeling can help to favor one program version over many others. For distributed memory organizations, the main difference of the parallel program versions is often the data distribution and the resulting communication requirements.

For different programming models, different challenges arise for the analytical modeling. For programming models with a distributed address space, communication and synchronization operations are called explicitly in the parallel program, which facilitates the performance modeling. The modeling can capture the actual communication times quite accurately, if the runtime of the single communication operations can be modeled quite accurately. This is typically the case for many execution platforms. For programming models with a shared address space, accesses to different memory locations may result in different access times, depending on the memory organization of the execution platform. Therefore, it is typically much more difficult to analytically capture the access time caused by a memory access. In the following, we consider programming models with a distributed address space.

The time for the execution of local computations can often be estimated by the number of (arithmetical or logical) operations to be performed. But there are several sources of inaccuracy that must be taken into consideration:

- It may not be possible to determine the number of arithmetical operations exactly, since loop bounds may not be known at compile time or since adaptive features are included to adapt the operations to a specific input situation. Therefore, for some operations or statements, the frequency of execution may not be known. Different approaches can be used to support analytical modeling in such situations. One approach is that the programmer can give hints in the program about the estimated number of iterations of a loop or the likelihood of a condition to be true or false. These hints can be included by pragma statements and could then be processed by a modeling tool.
  Another possibility is the use of profiling tools with which typical numbers of loop iterations can be determined for similar or smaller input sets. This information can then be used for the modeling of the execution time for larger input sets, e.g., using extrapolation.
- For different execution platforms, arithmetical operations may have distinct execution times, depending on their internal implementation. Larger differences may occur for more complex operations like division, square root, or trigonometric functions. However, these operations are not used very often. If larger differences occur, a differentiation between the operations can help for a more precise performance modeling.
- Each processor typically has a local memory hierarchy with several levels of caches. This results in varying memory access times for different memory locations. For the modeling, average access times can be used, computed from cache miss and cache hit rates, see Sect. 4.1.3. These rates can be obtained by profiling.

The time for data exchange between processors can be modeled by considering the communication operations executed during program execution in isolation. For a theoretical analysis of communication operations, asymptotic running times can be used. We consider these for different interconnection networks in the following.

4.3.1 Implementing Global Communication Operations

In this section, we study the implementation and asymptotic running times of various global communication operations introduced in Sect. 3.5.2 on static interconnection networks according to [19]. Specifically, we consider the linear array, the ring, a symmetric mesh, and the hypercube, as defined in Sect. 2.5.2. The parallel execution of global communication operations depends on the number of processors and the message size. The parallel execution time also depends on the topology of the network and the properties of the hardware realization. For the analysis, we make the following assumptions about the links and input and output ports of the network.

1. The links of the network are bidirectional, i.e., messages can be sent simultaneously in both directions. For real parallel systems, this property is usually fulfilled.

2. Each node can simultaneously send out messages on all its outgoing links; this is also called **all-port communication**. For parallel computers this can be organized by separate output buffers for each outgoing link of a node with corresponding controllers responsible for the transmission along that link. The simultaneous sending results from controllers working in parallel.
3. Each node can simultaneously receive messages on all its incoming links. In practice, there is a separate input buffer with controllers for each incoming link responsible for the receipt of messages.
4. Each message consists of several bytes, which are transmitted along a link without any interruption.
5. The time for transmitting a message consists of the startup time t_S, which is independent of the message size, and the byte transfer time $m \cdot t_B$, which is proportional to the size of the message m. The time for transmitting a single byte is denoted by t_B. Thus, the time for sending a message of size m from a node to a directly connected neighbor node takes time $T(m) = t_S + m \cdot t_B$, see also Formula (2.3) in Sect. 2.6.3.
6. Packet switching with store-and-forward is used as switching strategy, see also Sect. 2.6.3. The message is transmitted along a path in the network from the source node to a target node, and the length of the path determines the number of **time steps** of the transmission. Thus, the time for a communication also depends on the path length and the number of processors involved.

Given an interconnection network with these properties and parameters t_S and t_B, the time for a communication is mainly determined by the message size m and the path length p. For an implementation of global communication operations, several messages have to be transmitted and several paths are involved. For an efficient implementation, these paths should be planned carefully such that no conflicts occur. A conflict can occur when two messages are to be sent along the same link in the same time step; this usually leads to a delay of one of the messages, since the messages have to be sent one after another. Careful planning of the communication paths is a crucial point in the following implementation of global communication operations and the estimations of their running times. The execution times are given as asymptotic running time, which we briefly summarize now.

4.3.1.1 Asymptotic Notation

Asymptotic running times describe how the execution time of an algorithm increases with the size of the input, see, e.g., [31]. The notation for the asymptotic running time uses functions whose domains are the natural numbers $\mathbb{N}$. The function describes the essential terms for the asymptotic behavior and ignores less important terms such as constants and terms of lower increase. The asymptotic notation comprises the O-notation, the Ω-notation, and the Θ-notation, which describe boundaries of the increase of the running time. The asymptotic upper bound is given by the O-notation:

$$O(g(n)) = \{f(n) \mid \text{there exists a positive constant } c \text{ and } n_0 \in \mathbb{N}, \text{ such that for all } n \geq n_0 : 0 \leq f(n) \leq cg(n)\}.$$

The asymptotic lower bound is given by the Ω-notation:

$$\Omega(g(n)) = \{f(n) \mid \text{there exists a positive constant } c \text{ and } n_0 \in \mathbb{N}, \text{ such that for all } n \geq n_0 : 0 \leq cg(n) \leq f(n)\}.$$

The Θ-notation bounds the function from above and below:

$$\Theta(g(n)) = \{f(n) \mid \text{there exist positive constants } c_1, c_2 \text{ and } n_0 \in \mathbb{N}, \text{ such that for all } n \geq n_0 : 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)\}.$$

Figure 4.1 illustrates the boundaries for the O-notation, the Ω-notation, and the Θ-notation according to [31].

The asymptotic running times of global communication operations with respect to the number of processors in the static interconnection network are given in Table 4.1. Running times for global communication operations are presented often in the literature, see, e.g., [100, 75]. The analysis of running times mainly differs in the assumptions made about the interconnection network. In [75], one-port communication is considered, i.e., a node can send out only one message at a specific time step along one of its output ports; the communication times are given as functions in closed form depending on the number of processors p and the message size m for store-and-forward as well as cut-through switching. Here we use the assumptions given above according to [19].

The analysis uses the duality and hierarchy properties of global communication operation given in Fig. 3.9 in Sect. 3.5.2. Thus, from the asymptotic running times of one of the global communication operations it follows that a global communication operation which is less complex can be solved in no additional time and that a global communication operation which is more complex cannot be solved faster. For example, the scatter operation is less expensive than a multi-broadcast on the same network, but more expensive than a single-broadcast operation. Also a global communication operation has the same asymptotic time as its dual operation in the

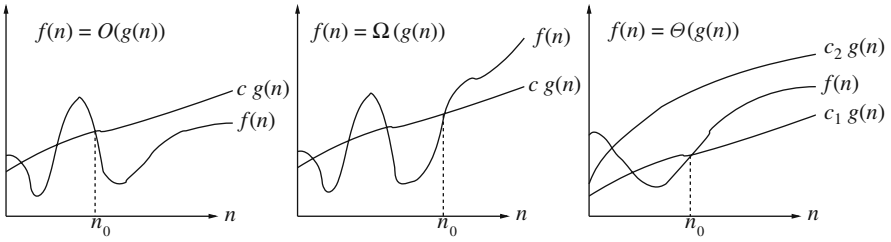


Fig. 4.1 Graphic examples of the O-, Ω-, and Θ-notation. As value for n_0 the minimal value which can be used in the definition is shown

Table 4.1 Asymptotic running times of the implementation of global communication operations depending on the number p of processors in the static network. The linear array has the same asymptotic times as the ring

Operation	Ring	Mesh	Hypercube
Single-broadcast	$\Theta(p)$	$\Theta(\sqrt[d]{p})$	$\Theta(\log p)$
Scatter	$\Theta(p)$	$\Theta(p)$	$\Theta(p/\log p)$
Multi-broadcast	$\Theta(p)$	$\Theta(p)$	$\Theta(p/\log p)$
Total exchange	$\Theta(p^2)$	$\Theta(p^{(d+1)/d})$	$\Theta(p)$

hierarchy. For example, the asymptotic time derived for a scatter operation can be used as asymptotic time of the gather operation.

4.3.1.2 Complete Graph

A complete graph has a direct link between every pair of nodes. With the assumption of bidirectional links and a simultaneous sending and receiving of each output port, a total exchange can be implemented in one time step. Thus, all other communication operations such as broadcast, scatter, and gather operations can also be implemented in one time step and the asymptotic time is $\Theta(1)$.

4.3.1.3 Linear Array

A linear array with p nodes is represented by a graph $G = (V, E)$ with a set of nodes $V = \{1, \ldots, p\}$ and a set of edges $E = \{(i, i+1) | 1 \leq i < p\}$, i.e., each node except the first and the final is connected with its left and right neighbors. For an implementation of a **single-broadcast operation**, the root processor sends the message to its left and its right neighbors in the first step; in the next steps each processor sends the message received from a neighbor in the previous step to its other neighbor. The number of steps depends on the position of the root processor. For a root processor at the end of the linear array, the number of steps is $p - 1$. For a root processor in the middle of the array, the time is $\lfloor p/2 \rfloor$. Since the diameter of a linear array is $p - 1$, the implementation cannot be faster and the asymptotic time $\Theta(p)$ results.

A **multi-broadcast operation** can also be implemented in $p-1$ time steps using the following algorithm. In the first step, each node sends its message to both neighbors. In the step $k = 2, \ldots, p-1$, each node i with $k \leq i < p$ sends the message received in the previous step from its left neighbor to the right neighbor $i + 1$; this is the message originating from node $i - k + 1$. Simultaneously, each node i with $2 \leq i \leq p - k + 1$ sends the message received in the previous step from its right neighbor to the left neighbor $i - 1$; this is the message originally coming from node $i + k - 1$. Thus, the messages sent to the right make one hop to the right per time step and the messages sent to the left make one hop to the left in one time step. After $p - 1$ steps, all messages are received by all nodes. Figure 4.2 shows a linear array with four nodes as example; a multi-broadcast operation on this linear array can be performed in three time steps.

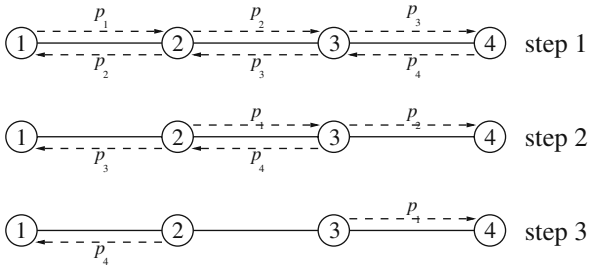


Fig. 4.2 Implementation of a multi-broadcast operation in time 3 on a linear array with four nodes

For the **scatter operation** on a linear array with p nodes, the asymptotic time $\Theta(p)$ results. Since the scatter operation is a specialization of the multi-broadcast operation it needs at most $p-1$ steps, and since the scatter operation is more general than a single-broadcast operation, it needs at least $p-1$ steps, see also the hierarchy of global communication operations in Fig. 3.9. When the root node of the scatter operation is not one of the end nodes of the array, a scatter operation can be faster. The messages for more distant nodes are sent out earlier from the root node, i.e., the messages are sent in the reverse order of their distance from the root node. All other nodes send the messages received in one step from one neighbor to the other neighbor in the next step.

The number of time steps for a **total exchange** can be determined by considering an edge $(k, k+1)$, $1 \leq k < p$, which separates the linear array into two subsets with k and $p-k$ nodes. Each node of the subset $\{1, \ldots, k\}$ sends $p-k$ messages along this edge to the other subset and each node of the subset $\{k+1, \ldots, p\}$ sends k messages in the other direction along this link. Thus, a total exchange needs at least $k \cdot (p-k)$ time steps or $p^2/4$ for $k = \lfloor p/2 \rfloor$. On the other hand, a total exchange can be implemented by p consecutive scatter operations, which lead to p^2 steps. Altogether, an asymptotic time $\Theta(p^2)$ results.

4.3.1.4 Ring

A ring topology has the nodes and edges of a linear array and an additional edge between node 1 and node p. All implementations of global communication operations are similar to the implementations on the linear array, but take one half of the time due to this additional link.

A **single-broadcast operation** is implemented by sending the message from the root node in both directions in the first step; in the following steps each node sends the message received in the opposite direction. This results in $\lfloor p/2 \rfloor$ time steps. Since the diameter of the ring is $\lceil p/2 \rceil$, the broadcast operation cannot be implemented faster and the time $\Theta(p)$ results.

A **multi-broadcast operation** is also implemented as for the array but in $\lfloor p/2 \rfloor$ steps. In the first step, each processor sends its message in both directions. In the following steps k, $2 \leq k \leq \lfloor p/2 \rfloor$, each processor sends the messages received in the opposite directions. Since the diameter is $\lceil p/2 \rceil$, the time $\Theta(p)$ results. Figure 4.3 illustrates a multi-broadcast operation for $p = 6$ processors.

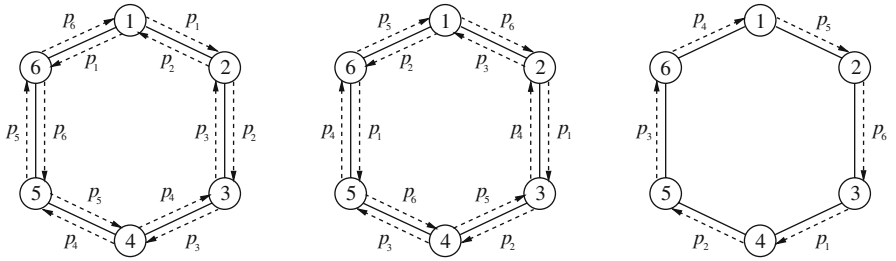


Fig. 4.3 Implementation of a multi-broadcast operation on a ring with six nodes. The message sent out by node i is denoted by p_i, $i = 1, \ldots, 6$

The **scatter operation** also needs time $\Theta(p)$ since it cannot be faster than a single-broadcast operation and it is not slower than a multi-broadcast operation. For a **total exchange**, the ring is divided into two sets of $p/2$ nodes each (for p even). Each node of one of the subsets sends $p/2$ messages into the other subset across two links. This results in $p^2/8$ time steps, since one message needs one time step to be sent along one link. The time is $\Theta(p^2)$.

4.3.1.5 Mesh

For a d-dimensional mesh with p nodes and $\sqrt[d]{p}$ nodes in each dimension, the diameter is $d(p^{1/d} - 1)$ and, thus, a single-broadcast operation can be executed in time $\Theta(p^{1/d})$. For the **scatter operation**, an upper bound is $\Theta(p)$ since a linear array with p nodes can be embedded into the mesh and a scatter operation needs time p on the array. A scatter operation also needs at least time $p-1$, since $p-1$ messages have to be sent along the d outgoing links of the root node, which takes $\lceil \frac{p-1}{d} \rceil$ time steps. The time $\Theta(p)$ for the **multi-broadcast operation** results in a similar way.

For the **total exchange**, we consider a mesh with an even number of nodes and subdivide the mesh into two submeshes of dimension $d-1$ with $p/2$ nodes each. Each node of a submesh sends $p/2$ messages into the other submesh, which have to be sent over the links connecting both submeshes. These are $(\sqrt[d]{p})^{d-1}$ links. Thus, at least $p^{\frac{d+1}{d}}$ time steps are needed (because of $p^2/(4p^{\frac{d-1}{d}}) = 1/(4p^{\frac{d-1-2d}{d}}) = \frac{1}{4}p^{\frac{d+1}{d}}$).

To show that a total exchange can be performed in time $O(p^{\frac{d+1}{d}})$, we consider an algorithm implementing the total exchange in time $p^{\frac{d+1}{d}}$. Such an algorithm can

be defined inductively from total exchange operations on meshes with lower dimension. For $d = 1$, the mesh is identical to a linear array for which the total exchange has a time complexity $O(p^2)$. Now we assume that an implementation on a $(d-1)$-dimensional symmetric mesh with time $O(p^{\frac{d}{d-1}})$ is given. The total exchange operation on the d-dimensional symmetric mesh can be executed in two phases. The d-dimensional symmetric mesh is subdivided into disjoint meshes of dimension $d-1$ which results in $\sqrt[d]{p}$ meshes. This can be done by fixing the value for the component in the last dimension x_d of the nodes $(x_1, \ldots, x_d)$ to one of the values $x_d = 1, \ldots, \sqrt[d]{p}$. In the first phase, total exchange operations are performed on the $(d-1)$-dimensional meshes in parallel. Since each $(d-1)$-dimensional mesh has $p^{\frac{d-1}{d}}$ nodes, in one of the total exchange operations $p^{\frac{d-1}{d}}$ messages are exchanged. Since p messages have to be exchanged in each $d-1$-dimensional mesh, there are $\frac{p}{p^{\frac{d-1}{d}}} = p^{1/d}$ total exchange operations to perform. Because of the induction hypothesis, each of the total exchange operations needs time $O\,p^{\frac{d-1}{d}\frac{d}{d-1}} = O(p)$ and thus the time $p^{1/d} \cdot O(p) = O(p^{\frac{d+1}{d}})$ for the first phase results. In the second phase, the messages between the different submeshes are exchanged. The d-dimensional mesh consists of $p^{\frac{d-1}{d}}$ meshes of dimension 1 with $\sqrt[d]{p}$ nodes each; these are linear arrays of size $\sqrt[d]{p}$. Each node of a one-dimensional mesh belongs to a different $d-1$-dimensional mesh and has already received $p^{\frac{d-1}{d}}$ messages in the first phase. Thus, each node of a one-dimensional mesh has $p^{\frac{d-1}{d}}$ messages different from the messages of the other nodes; these messages have to be exchanged between them. This takes time $O((\sqrt[d]{p})^2)$ for one message of each node and in total $p^{\frac{2}{d}} p^{\frac{d-1}{d}} = p^{\frac{d+1}{d}}$ time steps. Thus, the time complexity $\Theta(p^{\frac{d+1}{d}})$ results.

4.3.2 Communications Operations on a Hypercube

For a d-dimensional hypercube, we use the bit notation of the $p = 2^d$ nodes as d-bit words $\alpha = \alpha_1 \cdots \alpha_d \in \{0, 1\}^d$ introduced in Sect. 2.5.2.

4.3.2.1 Single-Broadcast Operation

A single-broadcast operation can be implemented using a spanning tree rooted at a node α that is the root of the broadcast operation. We construct a spanning tree for $\alpha = 00 \cdots 0 = 0^d$ and then derive spanning trees for other root nodes. Starting with root node $\alpha = 00 \cdots 0 = 0^d$ the children of a node are chosen by inverting one of the zero bits that are right of the rightmost unity bit. For $d = 4$ the spanning tree in Fig. 4.4 results.

The spanning tree with root $\alpha = 00 \cdots 0 = 0^d$ has the following properties: The bit names of two nodes connected by an edge differ in exactly one bit, i.e., the edges of the spanning tree correspond to hypercube links. The construction of the

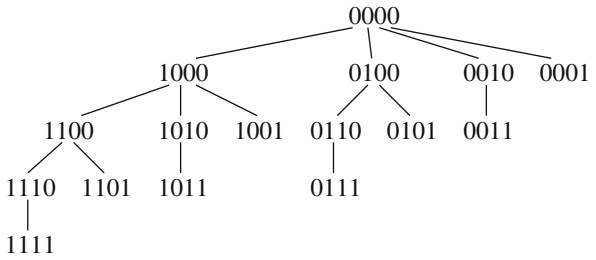


Fig. 4.4 Spanning tree for a single-broadcast operation on a hypercube for $d = 4$

spanning tree creates all nodes of the hypercube. All leaf nodes end with a unity. The maximal degree of a node is d, since at most d bits can be inverted. Since a child node has one more unity bit than its parent node, an arbitrary path from the root to a leaf has a length not larger than d, i.e., the spanning tree has depth d, since there is one path from the root to node $11\cdots1$ for which all d bits have to be inverted.

For a single-broadcast operation with an arbitrary root node z, a spanning tree T_z is constructed from the spanning tree T_0 rooted at node $00\cdots0$ by keeping the structure of the tree but mapping the bit names of the nodes to new bit names in the following way. A node x of tree T_0 is mapped to node $x \oplus z$ of tree T_z, where $\oplus$ denotes the bitwise `xor` operation (exclusive `or` operation), i.e.,

$$a_1 \cdots a_d \oplus b_1 \cdots b_d = c_1 \cdots c_d \text{ with } c_i = \begin{cases} 1 \text{ when } a_i \neq b_i \\ 0 \text{ otherwise} \end{cases} \quad \text{for } 1 \leq i \leq d.$$

Especially, node $\alpha = 00\cdots0$ is mapped to node $\alpha \oplus z = z$. The tree structure of tree T_z remains the same as for tree T_0. Since the nodes v, w of T_0 connected by an edge (v, w) differ in exactly one bit position, the nodes $v \oplus z$ and $w \oplus z$ of tree T_z also differ in exactly one bit position and the edge $(v \oplus z, w \oplus z)$ is a hypercube link. Thus, a spanning tree of the d-dimensional hypercube with root z results.

The spanning tree can be used to implement a single-broadcast operation from the root node in d time steps. The messages are first sent from the root to all children, and in the next time steps each node sends the message received to all its children. Since the diameter of a d-dimensional hypercube is d, the single-broadcast operation cannot be faster than d and the time $\Theta(d) = \Theta(\log(p))$ results.

4.3.2.2 Multi-broadcast Operation on a Hypercube

For a multi-broadcast operation, each node receives $p - 1$ messages from the other nodes. Since a node has $d = \log p$ incoming edges, which can receive messages simultaneously, an implementation of a multi-broadcast operation on a

d-dimensional hypercube takes at least $\lceil (p-1)/\log p \rceil$ time steps. There are algorithms that attain this lower bound and we construct one of them in the following according to [19].

The multi-broadcast operation is considered as a set of single-broadcast operations, one for each node in the hypercube. A spanning tree is constructed for the single-broadcast operations and the message is sent along the links of the tree in a sequence of time steps as described above for the single-broadcast in isolation. The idea of the algorithm for the multi-broadcast operation is to construct spanning trees for the single-broadcast operation such that the single-broadcast operations can be performed simultaneously. To achieve this, the links of the different spanning trees used for a transmission in the same time step have to be disjoint. This is the reason why the spanning trees for the single-broadcast in isolation cannot be used here as will be seen later. We start by constructing the spanning tree T_0 for root node $00\cdots0$.

The spanning tree T_0 for root node $00\cdots0$ consists of disjoint sets of edges $A_1, \ldots, A_m$, where m is the number of time steps needed for a single-broadcast and A_i is the set of edges over which the messages are transmitted at time step i, $i = 1, \ldots, m$. The set of start nodes of the edges in A_i is denoted by S_i and the set of end nodes is denoted by E_i, $i = 1, \ldots, m$, with $S_1 = \{(00\cdots0)\}$ and $S_i \subset S_1 \cup \bigcup_{k=1}^{i-1} E_k$. The spanning tree T_t with root $t \in \{0,1\}^d$ is constructed from T_0 by mapping the edge sets of T_0 to edge sets $A_i(t)$ of T_t using the `xor` operation, i.e.,

$$A_i(t) = \{(x \oplus t, y \oplus t) | (x, y) \in A_i\} \quad \text{for } 1 \leq i \leq m \ . \tag{4.9}$$

If T_0 is a spanning tree, then T_t is also a spanning tree with root $T \in \{0,1\}^d$. The goal is to construct the sets $A_1, \ldots, A_m$ such that for each $i \in \{1, \ldots, m\}$ the sets $A_i(t)$ are pairwise disjoint for all $t \in \{0,1\}^d$ (with $A_i = A_i(0)$, $i = 1, \ldots, m$). This means that transmission of data can be performed simultaneously on those links. To get disjoint edges for the same transmission step i, the sets A_i are constructed such that

– For any two edges $(x, y) \in A_i$ and $(x', y') \in A_i$, the bit position in which the nodes x and y differ is **not** the same bit position in which the nodes x' and y' differ.

The reason for this requirement is that two edges whose start and end nodes differ in the same bit position can be mapped onto each other by the `xor` operation with an appropriate t. Thus, if such edges would be in set A_i for some $i \in \{1, \ldots, m\}$, then they would be in the set $A_i(t)$ and the sets A_i and $A_i(t)$ would not be disjoint. This is illustrated in Fig. 4.5 for $d = 3$ using the spanning trees constructed earlier for the single-broadcast operations in isolation.

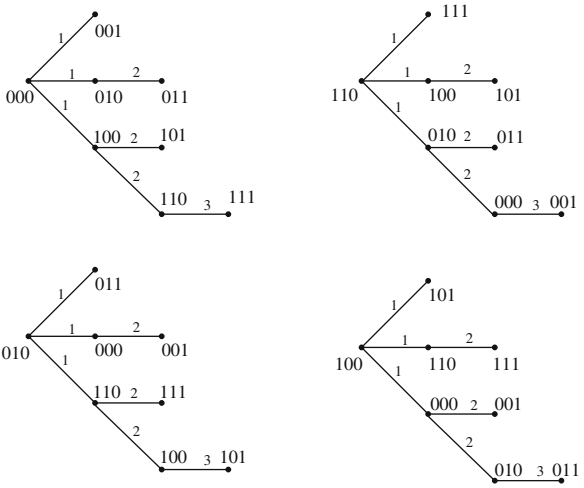


Fig. 4.5 Spanning tree for the single-broadcast operation in isolation. The start and end nodes of the edges $e_1 = ((010), (011))$ and $e_2 = ((100), (101))$ differ in the same bit position, which is the first bit position on the right. The `xor` operation with new root node $t = 110$ creates a tree that contains the same edges e_1 and e_2 for a data transmission in the second time step. A delay of the transmission into the third time step would solve this conflict. However, a new conflict in time step 3 results in the spanning tree with root 010, which has edge e_2 in the third time step, and in spanning tree with root 100, which has edge e_1 in the third time step

There are only d different bit positions so that each set A_i, $i = 1, \ldots, m$, can only contain at most d edges. Thus, the sets A_i are constructed such that $|A_i| = d$ for $1 \leq i < m$ and $|A_m| \leq d$. Since the sets $A_1, \ldots, A_m$ should be pairwise disjoint and the total number of edges in the spanning tree is $2^d - 1$ (there is an incoming edge for each node except the root node), we get

$$\left| \bigcup_{i=1}^{m} A_i \right| = 2^d - 1$$

and a first estimation for m:

$$m = \left\lceil \frac{2^d - 1}{d} \right\rceil .$$

Figure 4.6 shows the eight spanning trees for $d = 3$ and edge sets A_1, A_2, A_3 with $|A_1| = |A_2| = 3$ and $|A_3| = 1$. In this example, there is no conflict in any of the three time steps $i = 1, 2, 3$. These spanning trees can be used simultaneously, and a multi-broadcast needs $m = \lceil (2^3 - 1)/3 \rceil = 3$ time steps.

We now construct the edge sets A_i, $i = 1, \ldots, m$, for arbitrary d. The construction mainly consists of the following arrangement of the nodes of the d-dimensional

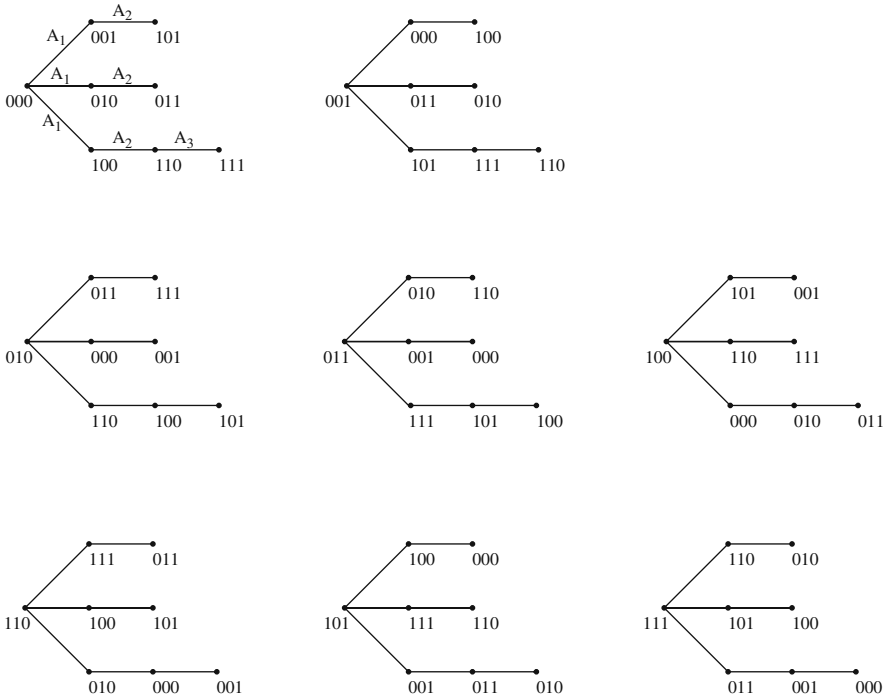


Fig. 4.6 Spanning trees for a multi-broadcast operation on a d-dimensional hypercube with $d = 3$. The sets A_1, A_2, A_3 for root 000 are $A_1 = \{(000, 001), (000, 010), (000, 100)\}$, $A_2 = \{(001, 101), (010, 011), (100, 110)\}$, and $A_3 = \{(110, 111)\}$ shown in the *upper left* corner. The other trees are constructed according to Formula (4.9)

hypercube. The set of nodes with k unity bits and $d - k$ zero bits is denoted as N_k, $k = 1, \ldots, d$, i.e.,

$$N_k = \{t \in \{0, 1\}^d \mid t \text{ has } k \text{ unity bits and } d - k \text{ zero bits}\}$$

for $0 \leq k \leq d$ with $N_0 = \{(00 \cdots 0)\}$ and $N_d = \{(11 \cdots 1)\}$. The number of elements in N_k is

$$|N_k| = \binom{d}{k} = \frac{d!}{k!(d-k)!} \, .$$

Each set N_k is further partitioned into disjoint sets $R_{k1}, \ldots, R_{kn_k}$, where one set R_{ki} contains all elements which result from a bit rotation to the left from each other. The sets R_{ki} are equivalence classes with respect to the relation *rotation to the left*. The first of these equivalence classes R_{k1} is chosen to be the set with the element $(0^{d-k}1^k)$, i.e., the rightmost bits are unity bits. Based on these sets, each node $t \in \{0, 1\}^d$ is assigned a number $n(t) \in \{0, \ldots, 2^d - 1\}$ corresponding to its position in the order

$$\{\alpha\} R_{11} R_{21} \cdots R_{2n_2} \cdots R_{k1} \cdots R_{kn_k} \cdots R_{(d-2)1} \cdots R_{(d-2)n_{d-2}} R_{(d-1)1} \{\beta\}, \quad (4.10)$$

with $\alpha = 00\cdots0$ and $\beta = 11\cdots1$ and position numbers $n(\alpha) = 0$ and $n(\beta) = 2^d - 1$. Each node $t \in \{0,1\}^d$, except α, is also assigned a number $m(t)$ with

$$m(t) = 1 + [(n(t) - 1) \bmod d], \quad (4.11)$$

i.e., the nodes are numbered in a round-robin fashion by $1, \ldots, d$. So far, there is no specific order of the nodes within one of the equivalence classes R_{kj}, $k = 1, \ldots, d$, $j = 1, \ldots, n_k$. Using $m(t)$ we now specify the following order:

– The first element $t \in R_{kj}$ is chosen such that the following condition is satisfied:

$$\text{The bit at position } m(t) \text{ from the right is 1.} \quad (4.12)$$

– The subsequent elements of R_{kj} result from a single bit rotation to the left. Thus, property (4.12) is satisfied for all elements of R_{kj}.

For the first equivalence classes R_{k1}, $k = 1, \ldots, d$, we additionally require the following:

– The first element $t \in R_{k1}$ has a zero at the bit position right of position $m(t)$, i.e., when $m(t) > 1$, the bit at position $m(t) - 1$ is a zero, and when $m(t) = 1$, the bit at the leftmost position is a zero.
– The property holds for all elements in R_{k1}, since they result by a bit rotation to the left from the first element.

For the case $d = 4$, the following order of the nodes $t \in \{0,1\}^4$ and $m(t)$ values result:

$$\begin{array}{ll}
N_0 & \overset{0}{(0000)} \\
N_1 & \underbrace{\overset{1}{(0001)} \quad \overset{2}{(0010)} \quad \overset{3}{(0100)} \quad \overset{4}{(1000)}}_{R_{11}} \\
N_2 & \underbrace{\overset{1}{(0011)} \quad \overset{2}{(0110)} \quad \overset{3}{(1100)} \quad \overset{4}{(1001)}}_{R_{21}} \quad \underbrace{\overset{1}{(0101)} \quad \overset{2}{(1010)}}_{R_{22}} \\
N_3 & \underbrace{\overset{3}{(1101)} \quad \overset{4}{(1011)} \quad \overset{1}{(0111)} \quad \overset{2}{(1110)}}_{R_{31}} \\
N_4 & \overset{3}{(1111)}\,.
\end{array}$$

Using the numbering $n(t)$ we now define the sets of end nodes $E_0, E_1, \ldots, E_m$ of the edge sets $A_1, \ldots, A_m$ as contiguous blocks of d nodes (or $< d$ nodes for the last set):

$$E_0 = \{(00\cdots0)\},$$
$$E_i = \{t \in \{0,1\}^d \mid (i-1)d + 1 \le n(t) \le i \cdot d\} \quad \text{for } 1 \le i < m,$$
$$E_m = \{t \in \{0,1\}^d \mid (m-1)d + 1 \le n(t) \le 2^d - 1\} \quad \text{with } m = \left\lceil \frac{2^d - 1}{d} \right\rceil.$$

The sets of edges A_i, $1 \le i \le m$, are then constructed according to the following:

- The set of edges A_i, $1 \le i \le m$, consists of the edges that connect an end node $t \in E_i$ with the start node t' obtained from t by inverting the bit at position $m(t)$, which is always a unity bit due to the construction.
- As an exception, the end node $t = (11\cdots1)$ for the case $m(11\cdots1) = d$ is connected to the start node $t' = (1011\cdots1)$ (and not $(011\cdots1)$).

Due to the construction the start nodes t' have one unity bit less than t and, thus, when $t \in N_k$, then $t' \in N_{k-1}$. Also the edges are links of the hypercube. Figure 4.7 shows the sets of end nodes and the sets of edges for $d = 4$.

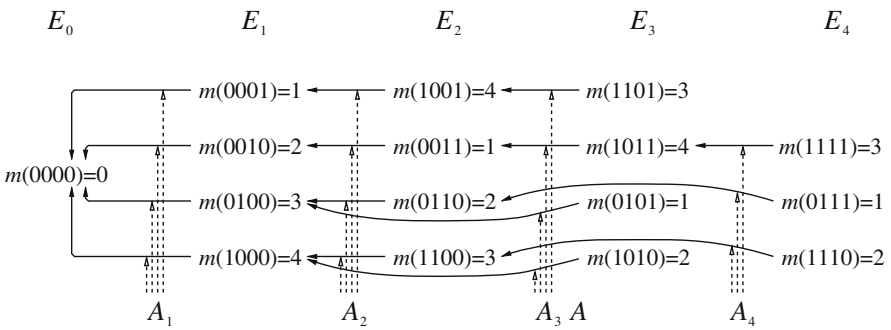


Fig. 4.7 Spanning tree with root node $00\cdots0$ for a multi-broadcast operation on a hypercube with $d = 4$. The sets of edges A_i, $i = 1, \ldots, 4$, are indicated by *dotted arrows*

Next, we show that these sets of edges define a spanning tree with root node $(00\cdots0)$ by showing that an end node $t \in E_i$ is connected to a start node $t' \in \bigcup_{k=1}^{i-1} E_k$, i.e., that there exists $k < i$ with $t' \in E_k$. Since t' has one more zero than t by construction, $n(t') < n(t)$ and thus $k > i$ is not possible, i.e., $k \le i$ holds. It remains to show that $k < i$.

- For $t = 11\cdots1$ and $m(t) = d$, the set E_m contains d nodes, which are node t and $d-1$ other nodes from $R_{d-1,1}$. There is one node of $R_{d-1,1}$ left, which is in set E_{m-1}; this node has a 1 at position $m(t)$ from the right and a 0 left of it. Thus, this node is $(1011\cdots1)$ which has been chosen as the start node by exception.
- For $t = 11\cdots1$ and $m(t) = d - k < d$, with $1 \le k < d$, the set E_m contains $d-k$ nodes s with numbers $n(s) < d-k$. The start node t' connected to t has a 0 at the position $d-k$ according to the construction and a 1 at the position $d-k-1$

from the right. Thus, $m(t') = d - k + 1$. Since $m(t') > d - k$, the node t' cannot belong to the edge set E_m and thus $t' \in E_{m-1}$.

For the nodes $t \neq 11\cdots1$, we now show that $n(t) - n(t') \geq d$, i.e., t' belongs to a different set E_k than t, with $k < i$.

- For $t \in R_{kn}$ with $n > 1$, all elements of R_{k1} are between t and t', since $t' \in N_{k-1}$. This set R_{k1} is the equivalence class of nodes $(0^{d-k}1^k)$ and contains d elements. Thus, $n(t) - n(t') \geq d$.
- For $t \in R_{k1}$, the start node t' is an element of $R_{k-1,1}$, since it has one more zero bit (which is at position $m(t)$) and according to the internal order in the set $R_{k-1,1}$ all remaining unity bits are right of $m(t)$ in a contiguous block of bit positions. Therefore, all elements of $R_{k-1,2}, \ldots, R_{k-1,n_{k-1}}$ are between t and t' . These are $|N_{k-1}| - |R_{k-1,1}| = \binom{d}{k-1} - d$ elements. For $2 < k < d$ and $d \geq 5$, it can be shown by induction that $\binom{d}{k-1} - d \geq d$. For $k = 1, 2$, $R_{11} = E_1$ and $R_{21} = E_2$ for all d and $t' \in E_{k-1}$ holds. For $d = 3$ and $d = 4$, the estimation can be shown individually; Fig. 4.6 shows the case $d = 3$ and Fig. 4.7 shows the case $d = 4$.

Thus, the sets $A_i(t)$, $i = 1, \ldots, m$, can be used for one of the single-broadcast operations of the multi-broadcast operation. The other sets $A_i(t)$ are constructed using the `xor` operation as described above. The trees can be used simultaneously, since no conflicts result. This can be seen from the construction and the numbers $m(t)$. The nodes in a set of end nodes E_i of edge set A_i have d different numbers $m(t) = 1, \ldots, d$ and, thus, for each of the nodes $t \in E_i$ a bit at a different bit position is inverted. Thus, the start and end nodes of the edges in A_i differ in different bit positions, which is the requirement to get a conflict-free transmission of messages in time step i. In summary, the single-broadcast operations can be performed in parallel and the multi-broadcast operation can be performed in $m = \lceil (2^d - 1)/d \rceil$ time steps.

4.3.2.3 Scatter Operation

A scatter operation takes no more time than the multi-broadcast operation, i.e., it takes no more than $\lceil (2^d - 1)/d \rceil$ time steps. On the other hand, in a scatter operation $2^d - 1$ messages have to be sent out from the d outgoing edges of the root node, which needs at least $\lceil (2^d - 1)/d \rceil$ time steps. Thus, the time for a scatter operation on a d-dimensional hypercube is $\Theta(\lceil (p-1)/\log p \rceil)$.

4.3.2.4 Total Exchange

The total exchange on a d-dimensional hypercube has time $\Theta(p) = \Theta(2^d)$. The lower bound results from decomposing the hypercube into two hypercubes of dimension $d - 1$ with $p/2 = 2^{d-1}$ nodes each and 2^{d-1} edges between them. For a total exchange, each node of one of the $(d - 1)$-dimensional hypercubes sends a

message for each node of the other hypercube; these are $(2^{d-1})^2 = 2^{2d-2}$ messages, which have to be transmitted along the 2^{d-1} edges connecting both hypercubes. This takes at least $2^{2d-2}/2^{d-1} = 2^{d-1} = p/2$ time steps.

An algorithm implementing the total exchange in $p-1$ steps can be built recursively. For $d = 1$, the hypercube consists of 2 nodes for which the total exchange can be done in one time step, which is $2^1 - 1$. Next, we assume that there is an implementation of the total exchange on a d-dimensional hypercube in time $\leq 2^d - 1$. A $(d+1)$-dimensional hypercube is decomposed into two hypercubes C_1 and C_2 of dimension d. The algorithm consists of the three phases:

1. A total exchange within the hypercubes C_1 and C_2 is performed simultaneously.
2. Each node in C_1 (or C_2) sends 2^d messages for the nodes in C_2 (or C_1) to its counterpart in the other hypercube. Since all nodes used different edges, this takes time 2^d.
3. A total exchange in each of the hypercubes is performed to distribute the messages received in phase 2.

The phases 1 and 2 can be performed simultaneously and take time 2^d. Phase 3 has to be performed after phase 2 and takes time $\leq 2^d - 1$. In summary, the time $2^d + 2^d - 1 = 2^{d+1} - 1$ results.

4.4 Analysis of Parallel Execution Times

The time needed for the parallel execution of a parallel program depends on

- the size of the input data n, and possibly further characteristics such as the number of iterations of an algorithm or the loop bounds;
- the number of processors p; and
- the communication parameters, which describe the specifics of the communication of a parallel system or a communication library.

For a specific parallel program, the time needed for the parallel execution can be described as a function $T(p,n)$ depending on p and n. This function can be used to analyze the parallel execution time and its behavior depending on p and n. As example, we consider the parallel implementations of a scalar product and of a matrix–vector product, presented in Sect. 3.6.

4.4.1 Parallel Scalar Product

The parallel scalar product of two vectors $\mathbf{a}, \mathbf{b} \in \mathbb{R}^n$ computes a scalar value which is the sum of the values $a_j \cdot b_j$, $j = 1, \ldots, n$. For a parallel computation on p processors, we assume that n is divisible by p with $n = r \cdot p$, $r \in \mathbb{N}$, and that the vectors are distributed in a blockwise way, see Sect. 3.4 for a description of data distributions. Processor P_k stores the elements a_j and b_j with $r \cdot (k-1)+1 \leq j \leq r \cdot k$ and computes the partial scalar products

$$c_k = \sum_{j=r\cdot(k-1)+1}^{r\cdot k} a_j \cdot b_j \ ,$$

so that processor P_k stores value c_k. To get the final result $c = \sum_{k=1}^{p} c_k$, a single-accumulation operation is performed and one of the processors stores this value. The parallel execution time of the implementation depends on the computation time and the communication time. To build a function $T(p, n)$, we assume that the execution of an arithmetic operation needs α time units and that sending a floating-point value to a neighboring processor in the interconnection network needs β time units. The parallel computation time for the partial scalar product is $2r\alpha$, since about r addition operations and r multiplication operations are performed.

The time for a single-accumulation operation depends on the specific interconnection network and we consider the linear array and the hypercube as examples. See also Sect. 2.5.2 for the definition of these direct networks.

4.4.1.1 Linear Array

In the linear array, the optimal processor as root node for the single-accumulation operation is the node in the middle since it has a distance no more than $p/2$ from every other node. Each node gets a value from its left (or right) neighbor in time β, adds the value to the local value in time α, and sends the results to its right (or left) in the next step. This results in the communication time $\frac{p}{2}(\alpha + \beta)$. In total, the parallel execution time is

$$T(p, n) = 2\frac{n}{p}\alpha + \frac{p}{2}(\alpha + \beta). \tag{4.13}$$

The function $T(p, n)$ shows that the computation time decreases with increasing number of processors p but that the communication time increases with increasing number of processors. Thus, this function exhibits the typical situation in a parallel program that an increasing number of processors does not necessarily lead to faster programs since the communication overhead increases. Usually, the parallel execution time decreases for increasing p until the influence of the communication overhead is too large and then the parallel execution time increases again. The value for p at which the parallel execution time starts to increase again is the optimal value for p, since more processors do not lead to a faster parallel program.

For Function (4.13), we determine the optimal value of p which minimizes the parallel execution time for $T(p) \equiv T(p, n)$ using the derivatives of this function. The first derivative is

$$T'(p) = -\frac{2n\alpha}{p^2} + \frac{\alpha + \beta}{2} \ ,$$

when considering $T(p)$ as a function of real values. For $T'(p) = 0$, we get $p^* = \pm\sqrt{\frac{4n\alpha}{\alpha+\beta}}$. The second derivative is $T''(p) = \frac{4n\alpha}{p^3}$ and $T''(p^*) > 0$, meaning that $T(p)$ has a minimum at p^*. From the formula for p^*, we see that the optimal number of processors increases with $\sqrt{n}$. We also see that $p^* = 2\sqrt{\frac{\alpha}{\alpha+\beta}}\sqrt{n} < 1$, if $\beta > (4n-1)\alpha$, so that the sequential program should be used in this case.

4.4.1.2 Hypercube

For the d-dimensional hypercube with $d = \log p$, the single-accumulation operation can be performed in $\log p$ time steps using a spanning tree, see Sect. 4.3.1. Again, each step for sending a data value to a neighboring node and the local addition takes time $\alpha + \beta$ so that the communication time $\log p(\alpha + \beta)$ results. In total, the parallel execution time is

$$T(n, p) = \frac{2n\alpha}{p} + \log p \cdot (\alpha + \beta) \; . \tag{4.14}$$

This function shows a slightly different behavior of the overhead than Function (4.13). The communication overhead increases with the factor $\log p$. The optimal number of processors is again determined by using the derivatives of $T(p) \equiv T(n, p)$. The first derivative (using $\log p = \ln p / \ln 2$ with the natural logarithm) is

$$T'(p) = -\frac{2n\alpha}{p^2} + (\alpha + \beta)\frac{1}{p}\frac{1}{\ln 2} \; .$$

For $T'(p) = 0$, we get the necessary condition $p^* = \frac{2n\alpha \ln 2}{\alpha+\beta}$. Since $T''(p) = \frac{4n\alpha}{p^3} - \frac{1}{p^2}\frac{\alpha+\beta}{ln2} > 0$ for p^*, the function $T(p)$ has a minimum at p^*. This shows that the optimal number of processors increases with increasing n. This is faster than for the linear array and is caused by the faster implementation of the single-accumulation operation.

4.4.2 Parallel Matrix–Vector Product

The parallel implementation of the matrix–vector product $\mathbf{A} \cdot \mathbf{b} = \mathbf{c}$ with $\mathbf{A} \in \mathbb{R}^{n \times n}$ and $\mathbf{b} \in \mathbb{R}^n$ can be performed with a row-oriented distribution of the matrix A or with column-oriented distribution of matrix A, see Sect. 3.6. For deriving a function describing the parallel execution time, we assume that n is a multiple of the number of processors p with $r = \frac{n}{p}$ and that an arithmetic operation needs α time units.

- For an implementation using a row-oriented distribution of blocks of rows, processor P_k stores the rows i with $r \cdot (k-1) + 1 \leq i \leq r \cdot k$ of matrix $\mathbf{A}$ and computes the elements

$$c_i = \sum_{j=1}^{n} a_{ij} \cdot b_j$$

of the result vector c. For each of these r values, the computation needs n multiplication and $n-1$ addition operations so that approximately the computation time $2nr\alpha$ is needed. The vector **b** is replicated for this computation. If the result vector **c** has to be replicated as well, a multi-broadcast operation is performed, for which each processor P_k, $k = 1, \ldots, p$, provides $r = \frac{n}{p}$ elements.

- For an implementation with column-oriented distribution of blocks of columns, processor P_k stores the columns j with $r \cdot (k-1)+1 \leq j \leq r \cdot k$ of matrix **A** as well as the corresponding elements of **b** and computes a partial linear combination, i.e., P_k computes n partial sums $d_{k1}, \ldots, d_{kn}$ with

$$d_{kj} = \sum_{l=r\cdot(k-1)+1}^{r\cdot k} a_{jl} b_l \ .$$

The computation of each d_{kj} needs r multiplications and $r-1$ additions so that for all n values the approximate computation time $n2r\alpha$ results. A final multi-accumulation operation with addition as reduction operation computes the final result **c**. Each processor P_k adds the values $d_{1j}, \ldots, d_{nj}$ for $(k-1) \cdot r + 1 \leq j \leq k \cdot r$, i.e., P_k performs an accumulation with blocks of size r and vector **c** results in a blockwise distribution.

Thus, both implementation variants have the same execution time $2\frac{n^2}{p}\alpha$. Also, the communication time is asymptotically identical, since multi-broadcast and multi-accumulation are dual operations, see Sect. 3.5. For determining a function for the communication time, we assume that sending r floating-point values to a neighboring processor in the interconnection network needs $\beta + r \cdot \gamma$ time units and consider the two networks, a linear array and a hypercube.

4.4.2.1 Linear Array

In the linear array with p processors, a multi-broadcast operation (or a multi-accumulation) operation can be performed in p steps in each of which messages of size r are sent. This leads to a communication time $p(\beta + r \cdot \gamma)$. Since the message size in this example is $r = \frac{n}{p}$, the following parallel execution time results:

$$T(n, p) = \frac{2n^2}{p}\alpha + p \cdot \left(\beta + \frac{n}{p} \cdot \gamma\right) = \frac{2n^2}{p}\alpha + p \cdot \beta + n \cdot \gamma \ .$$

This function shows that the computation time decreases with increasing p but the communication time increases linearly with increasing p, which is similar as for the scalar product. But in contrast to the scalar product, the computation time increases quadratically with the system size n, whereas the communication time increases

only linearly with the system size n. Thus, the relative communication overhead is smaller. Still, for a fixed number n, only a limited number of processors p leads to an increasing speedup.

To determine the optimal number p^* of processors, we again consider the derivatives of $T(p) \equiv T(n, p)$. The first derivative is

$$T'(p) = -\frac{2n^2\alpha}{p^2} + \beta \ ,$$

for which $T'(p) = 0$ leads to $p^* = \sqrt{2\alpha n^2/\beta} = n \cdot \sqrt{2\alpha/\beta}$. Since $T''(p) = 4\alpha n^2/p^3$, we get $T''(n\sqrt{2\alpha/\beta}) > 0$ so that p^* is a minimum of $T(p)$. This shows that the optimal number of processors increases linearly with n.

4.4.2.2 Hypercube

In a $\log p$-dimensional hypercube, a multi-broadcast (or a multi-accumulation) operation needs $p/\log p$ steps, see Sect. 4.3, with $\beta + r \cdot \gamma$ time units in each step. This leads to a parallel execution time:

$$\begin{aligned} T(n, p) &= \frac{2\alpha n^2}{p} + \frac{p}{\log p}(\beta + r \cdot \gamma) \\ &= \frac{2\alpha n^2}{p} + \frac{p}{\log p} \cdot \beta + \frac{\gamma n}{\log p} \ . \end{aligned}$$

The first derivative of $T(p) \equiv T(n, p)$ is

$$T'(p) = -\frac{2\alpha n^2}{p^2} + \frac{\beta}{\log p} - \frac{\beta}{\log^2 p \ln 2} - \frac{\gamma n}{p \cdot \log^2 p \ln 2} \ .$$

For $T'(p) = 0$ the equation

$$-2\alpha n^2 \log^2 p + \beta p^2 \log p - \beta p^2 \frac{1}{\ln 2} - \gamma n p \frac{1}{\ln 2} = 0$$

needs to be fulfilled. This equation cannot be solved analytically, so that the number of optimal processors p^* cannot be expressed in closed form. This is a typical situation for the analysis of functions for the parallel execution time, and approximations are used. In this specific case, the function for the linear array can be used since the hypercube can be embedded into a linear array. This means that the matrix–vector product on a hypercube is at least as fast as on the linear array.

4.5 Parallel Computational Models

A computational model of a computer system describes at an abstract level which basic operations can be performed when the corresponding actions take effect and how data elements can be accessed and stored [14]. This abstract description does not consider details of a hardware realization or a supporting runtime system. A computational model can be used to evaluate algorithms independently of an implementation in a specific programming language and of the use of a specific computer system. To be useful, a computational model must abstract from many details of a specific computer system while on the other hand it should capture those characteristics of a broad class of computer systems which have a larger influence on the execution time of algorithms.

To evaluate a specific algorithm in a computational model, its execution according to the computational model is considered and analyzed concerning a specific aspect of interest. This could, for example, be the number of operations that must be performed as a measure for the resulting execution time or the number of data elements that must be stored as a measure for the memory consumption, both in relation to the size of the input data. In the following, we give a short overview of popular parallel computational models, including the PRAM model, the BSP model, and the LogP model. More information on computational models can be found in [156].

4.5.1 PRAM Model

The theoretical analysis of sequential algorithms is often based on the RAM (Random Access Machine) model which captures the essential features of traditional sequential computers. The RAM model consists of a single processor and a memory with a sufficient capacity. Each memory location can be accessed in a random (direct) way. In each time step, the processor performs one instruction as specified by a sequential algorithm. Instructions for (read or write) access to the memory as well as for arithmetic or logical operations are provided. Thus, the RAM model provides a simple model which abstracts from many details of real computers, like a fixed memory size, existence of a memory hierarchy with caches, complex addressing modes, or multiple functional units. Nevertheless, the RAM model can be used to perform a runtime analysis of sequential algorithms to describe their asymptotic behavior, which is also meaningful for real sequential computers.

The RAM model has been extended to the PRAM (Parallel Random Access Machine) model to analyze parallel algorithms [53, 98, 123]. A PRAM consists of a bounded set of identical processors $\{P_1, \ldots, P_n\}$, which are controlled by a global clock. Each processor is a RAM and can access the common memory to read and write data. All processors execute the same program synchronously. Besides the common memory of unbounded size, there is a local memory for each processor to store private data. Each processor can access any location in the common memory

in unit time, which is the same time needed for an arithmetic operation. The PRAM executes computation steps one after another. In each step, each processor (a) reads data from the common memory or its private memory (read phase), (b) performs a local computation, and (c) writes a result back into the common memory or into its private memory (write phase). It is important to note that there is no direct connection between the processors. Instead, communication can only be performed via the common memory.

Since each processor can access any location in the common memory, memory access conflicts can occur when multiple processors access the same memory location at the same time. Such conflicts can occur in both the read phase and the write phase of a computation step. Depending on how these read conflicts and write conflicts are handled, several variants of the PRAM model are distinguished. The **EREW** (*exclusive read, exclusive write*) PRAM model forbids simultaneous read accesses as well as simultaneous write accesses to the same memory location by more than one processor. Thus, in each step, each processor must read from and write into a different memory location as the other processors. The **CREW** (*concurrent read, exclusive write*) PRAM model allows simultaneous read accesses by multiple processors to the same memory location in the same step, but simultaneous write accesses are forbidden within the same step. The **ERCW** (*exclusive read, concurrent write*) PRAM model allows simultaneous write accesses, but forbids simultaneous read accesses within the same step. The **CRCW** (*concurrent read, concurrent write*) PRAM model allows both simultaneous read and write accesses within the same step. If simultaneous write accesses are allowed, write conflicts to the same memory location must be resolved to determine what happens if multiple processors try to write to the same memory location in the same step. Different resolution schemes have been proposed:

(1) The *common model* requires that all processors writing simultaneously to a common location write the same value.
(2) The *arbitrary model* allows an arbitrary value to be written by each processor; if multiple processors simultaneously write to the same location, an arbitrarily chosen value will succeed.
(3) The *combining model* assumes that the values written simultaneously to the same memory location in the same step are combined by summing them up and the combined value is written.
(4) The *priority model* assigns priorities to the processors and in the case of simultaneous writes the processor with the highest priority succeeds.

In the PRAM model, the cost of an algorithm is defined as the number of PRAM steps to be performed for the execution of an algorithm. As described above, each step consists of a read phase, a local computation, and a write phase. Usually, the costs are specified as asymptotic execution time with respect to the size of the input data. The theoretical PRAM model has been used as a concept to build the SB-PRAM as a real parallel machine which behaves like the PRAM model [1, 101]. This machine is an example for simultaneous multi-threading, since the

unit memory access time has been reached by introducing logical processors which are simulated in a round-robin fashion and, thus, hide the memory latency.

A useful class of operations for PRAM models or PRAM-like machines is the multi-prefix operations which can be defined for different basic operations. We consider an MPADD operation as example. This operation works on a variable s in the common memory. The variable s is initialized to the value o. Each of the processors P_i, $i = 1, \ldots, n$, participating in the operation provides a value o_i. The operation is synchronously executed and has the effect that processor P_j obtains the value

$$o + \sum_{i=1}^{j-1} o_i .$$

After the operation, the variable s has the value $o + \sum_{i=1}^{n} o_i$. Multi-prefix operations can be used for the implementation of synchronization operations and parallel data structures that can be accessed by multiple processors simultaneously without causing race conditions [76]. For an efficient implementation, hardware support or even a hardware implementation for multi-prefix operations is useful as has been provided by the SB-PRAM prototype [1]. Multi-prefix operations are also useful for the implementation of a parallel task pool providing a dynamic load balancing for application programs with an irregular computational behavior, see [76, 102, 141, 149]. An example for such an application is the Cholesky factorization for sparse matrices for which the computational behavior depends on the sparsity structure of the matrix to be factorized. Section 7.5 gives a detailed description of this application. The implementation of task pools in Pthreads is considered in Sect. 6.1.6.

A theoretical runtime analysis based on the PRAM model provides useful information on the asymptotic behavior of parallel algorithms. But the PRAM model has its limitations concerning a realistic performance estimation of application programs on real parallel machines. One of the main reasons for these limitations is the assumption that each processor can access any location in the common memory in unit time. Real parallel machines do not provide memory access in unit time. Instead, large variations in memory access time often occur, and accesses to a global memory or to the local memory of other processors are usually much slower than accesses to the local memory of the accessing processor. Moreover, real parallel machines use a memory hierarchy with several levels of caches with different access times. This cannot be modeled with the PRAM model. Therefore, the PRAM model cannot be used to evaluate the locality behavior of the memory accesses of a parallel application program. Other unrealistic assumptions of the PRAM model are the synchronous execution of the processors and the absence of collisions when multiple processors access the common memory simultaneously. Because of these structures, several extensions of the original PRAM model have been proposed. The missing synchronicity of instruction execution in real parallel machines is addressed in the *phase PRAM* model [66], in which the computations are partitioned into phases such that the processors work asynchronously within the phases. At the end of each

phase, a barrier synchronization is performed. The *delay PRAM* model [136] tries to model delays in memory access times by introducing a communication delay between the time at which a data element is produced by a processor and the time at which another processor can use this data element. A similar approach is used for the *local memory PRAM* and the *block PRAM* model [4, 5]. For the block PRAM, each access to the common memory takes time $l + b$, where l is a startup time and b is the size of the memory block addressed. A more detailed description of PRAM models can be found in [29].

4.5.2 BSP Model

None of the PRAM models proposed has really been able to capture the behavior of real parallel machines for a large class of application areas in a satisfactory way. One of the reasons is that there is a large variety of different architectures for parallel machines and the architectures are steadily evolving. To avoid that the computational model design constantly drags behind the development of parallel computer architecture, the BSP model (*bulk synchronously parallel*) has been proposed as a bridging model between hardware architecture and software development [171]. The idea is to provide a standard on which both hardware architects and software developers can agree. Thus, software development can be decoupled from the details of a specific architecture, and software does not have to be adapted when porting it to a new parallel machine.

The BSP model is an abstraction of a parallel machine with a physically distributed memory organization. Communication between the processors is not performed as separate point-to-point transfers, but is bundled in a step-oriented way. In the BSP model, a parallel computer consists of a number of *components* (processors), each of which can perform processing or memory functions. The components are connected by a *router* (interconnection network) which can send point-to-point messages between pairs of components. There is also a *synchronization unit*, which supports the synchronization of all or a subset of the components. A computation in the BSP model consists of a sequence of *supersteps*, see Fig. 4.8 for an illustration. In each superstep, each component performs local computations and can participate in point-to-point message transmissions. A local computation can be performed in one time unit. The effect of message transmissions becomes visible in the next time step, i.e., a receiver of a message can use the received data not before the next superstep. At the end of each superstep, a **barrier synchronization** is performed. There is a *periodicity parameter* L which determines the length of the supersteps in time units. Thus, L determines the granularity of the computations. The BSP model allows that the value of L can be controlled by the program to be executed, even at runtime. There may be a lower bound for L given by the hardware. The parallel program to be executed should set an upper bound for L such that in each superstep, computations with approximately L steps can be assigned to each processor.

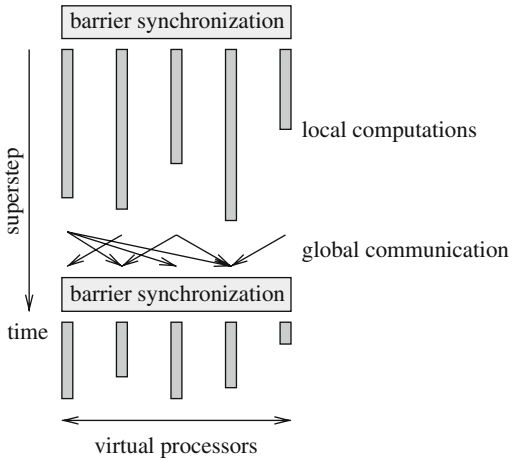


Fig. 4.8 In the BSP model, computations are performed in supersteps where each superstep consists of three phases: (1) simultaneous local computations of each processor, (2) communication operations for data exchange between processors, and (3) a barrier synchronization to terminate the communication operations and to make the data sent visible to the receiving processors. The communication pattern shown for the communication phase represents an h-relation with $h = 3$

In each superstep, the router can implement arbitrary *h-relations* capturing communication patterns, where each processor sends or receives at most h messages. A computation in the BSP model can be characterized by four parameters [89]:

- p: the number of (virtual) processors used within the supersteps to perform computations;
- s: the execution speed of the processors expressed as the number of computation steps per seconds that each processor can perform, where each computation step performs an (arithmetic or logical) operation on a local data element;
- l: the number of steps required for the execution of a barrier synchronization;
- g: the number of steps required on the average for the transfer of a memory word in the context of an h-relation.

The parameter g is determined such that the execution of an h-relation with m words per message takes $l \cdot m \cdot g$ steps. For a real parallel computer, the value of g depends not only on the bisection bandwidth of the interconnection network, see p. 30, but also on the communication protocol used and on the implementation of the communication library. The value of l is influenced not only by the diameter of the interconnection network, but also by the implementation of the communication library. Both l and g can be determined by suitable benchmark programs. Only p, l, and g are independent parameters; the value of s is used for the normalization of the values of l and g.

The execution time of a BSP program is specified as the sum of the execution times of the supersteps which are performed for executing the program. The execution time $T_{\text{superstep}}$ of a single superstep consists of three terms: (1) the maximum of the execution time w_i for performing local computations of processor P_i, (2) the

time for global communication for the implementation of an h-relation, and (3) the time for the barrier synchronization at the end of each superstep. This results in

$$T_{\text{superstep}} = \max_{\text{processors}} w_i + h \cdot g + l.$$

The BSP model is a general model that can be used as a basis for different programming models. To support the development of efficient parallel programs with the BSP model, the BSPLib library has been developed [74, 89], which provides operations for the initialization of a superstep, for performing communication operations, and for participating in the barrier synchronization at the end of each superstep.

The BSP model has been extended to the Multi-BSP model, which extends the original BSP model to capture important characteristics of modern architectures, in particular multicore architectures [172]. In particular, the model is extended to a hierarchical model with an arbitrary number d of levels modeling multiple memory and cache levels. Moreover, at each level the memory size is incorporated as an additional parameter. The entire model is based on a tree of depth d with memory/caches at the internal nodes and processors at the leaves.

4.5.3 LogP Model

In [34], several concerns about the BSP model are formulated. First, the length of the supersteps must be sufficiently large to accommodate arbitrary h-relations. This has the effect that the granularity cannot be decreased below a certain value. Second, messages sent within a superstep can only be used in the next superstep, even if the interconnection network is fast enough to deliver messages within the same superstep. Third, the BSP model expects hardware support for synchronization at the end of each superstep. Such support may not be available for some parallel machines. Because of these concerns, the BSP model has been extended to the *LogP model* to provide a more realistic modeling of real parallel machines.

Similar to the BSP model, the LogP model is based on the assumption that a parallel computer consists of a set of processors with local memory that can communicate by exchanging point-to-point messages over an interconnection network. Thus, the LogP model is also intended for the modeling of parallel computers with a distributed memory. The communication behavior of a parallel computer is described by four parameters:

- L (latency) is an upper bound on the latency of the network capturing the delay observed when transmitting a small message over the network;
- o (overhead) is the management overhead that a processor needs for sending or receiving a message; during this time, a processor cannot perform any other operation;
- g (gap) is the minimum time interval between consecutive send or receive operations of a processor;
- P (processors) is the number of processors of the parallel machine.

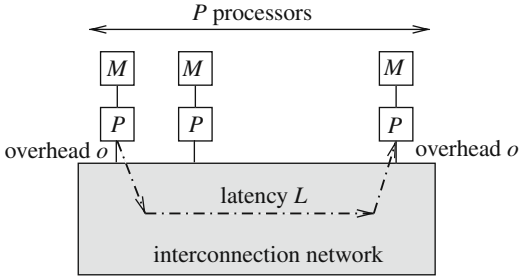


Fig. 4.9 Illustration of the parameters of the LogP model

Figure 4.9 illustrates the meaning of these parameters [33]. All parameters except P are measured in time units or as multiples of the machine cycle time. Furthermore it is assumed that the network has a *finite capacity* which means that between any pair of processors at most $\lceil L/g \rceil$ messages are allowed to be in transmission at any time. If a processor tries to send a message that would exceed this limit, it is blocked until the message can be transmitted without exceeding the limit. The LogP model assumes that the processors exchange *small messages* that do not exceed a predefined size. Larger messages must be split into several smaller messages. The processors work *asynchronously* with each other. The latency of any single message cannot be predicted in advance, but is bounded by L if there is no blocking because of the finite capacity. This includes that messages do not necessarily arrive in the same order in which they have been sent. The values of the parameters L, o, and g depend not only on the hardware characteristics of the network, but also on the communication library and its implementation.

The execution time of an algorithm in the LogP model is determined by the maximum of the execution times of the participating processors. An access by a processor P_1 to a data element that is stored in the local memory of another processor P_2 takes time $2 \cdot L + 4 \cdot o$; half of this time is needed to bring the data element from P_2 to P_1, the other half is needed to bring the data element from P_1 back to P_2. A sequence of n messages can be transmitted in time $L + 2 \cdot o + (n-1) \cdot g$, see Fig. 4.10.

A drawback of the original LogP model is that it is based on the assumption that the messages are small and that only point-to-point messages are allowed. More complex communication patterns must be assembled from point-to-point messages.

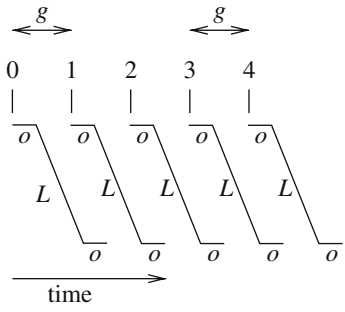


Fig. 4.10 Transmission of a larger message as a sequence of n smaller messages in the LogP model. The transmission of the last smaller message is started at time $(n-1) \cdot g$ and reaches its destination $2 \cdot o + L$ time units later

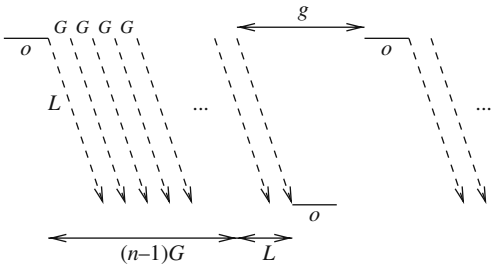


Fig. 4.11 Illustration of the transmission of a message with n bytes in the LogGP model. The transmission of the last byte of the message is started at time $o + (n - 1) \cdot G$ and reaches its destination $o + L$ time units later. Between the transmission of the last byte of a message and the start of the transmission of the next message at least g time units must have elapsed

To release the restriction to small messages, the LogP model has been extended to the LogGP model [10], which contains an additional parameter G (*Gap per byte*). This parameter specifies the transmission time per byte for long messages. $1/G$ is the bandwidth available per processor. The time for the transmission of a message with n bytes takes time $o + (n - 1)G + L + o$, see Fig. 4.11.

The LogGP model has been successfully used to analyze the performance of message-passing programs [9, 104]. The LogGP model has been further extended to the LogGPS model [96] by adding a parameter S to capture synchronization that must be performed when sending large messages. The parameter S is the threshold for the message length above which a synchronization between sender and receiver is performed before message transmission starts.

4.6 Exercises for Chap. 4

Exercise 4.1 We consider two processors P_1 and P_2 which have the same set of instructions. P_1 has a clock rate of 4 GHz, P_2 has a clock rate of 2 GHz. The instructions of the processors can be partitioned into three classes A, B, and C. The following table specifies for each class the CPI values for both processors. We assume that there are three compilers C_1, C_2, and C_3 available for both processors. We consider a specific program X. All three compilers generate machine programs which lead to the execution of the same number of instructions. But the instruction classes are represented with different proportions according to the following table:

Class	CPI for P_1	CPI for P_2	C_1 (%)	C_2 (%)	C_3 (%)
A	4	2	30	30	50
B	6	4	50	20	30
C	8	3	20	50	20

(a) If C_1 is used for both processors, how much faster is P_1 than P_2?
(b) If C_2 is used for both processors, how much faster is P_2 than P_2?

(c) Which of the three compilers is best for P_1?
(d) Which of the three compilers is best for P_2?

Exercise 4.2 Consider the MIPS (Million Instructions Per Second) rate for estimating the performance of computer systems for a computer with instructions $I_1, \ldots, I_m$. Let p_k be the proportion with which instruction $I_k (1 \leq k \leq m)$ is represented in the machine program for a specific program X with $0 \leq p_k \leq 1$. Let CPI_k be the CPI value for I_k and let t_c be the cycle time of the computer system in nanoseconds (10^{-9}).

(a) Show that the MIPS rate for program X can be expressed as

$$MIPS(X) = \frac{1000}{(p_1 \cdot CPI_1 + \cdots + p_m CPI_m) \cdot t_c[\text{ns}]}.$$

(b) Consider a computer with a clock rate of 3.3 GHz. The CPI values and proportion of occurrence of the different instructions for program X are given in the following table

Instruction I_k	p_n	CPI_n
Load and store	20.4	2.5
Integer add and subtract	18.0	1
Integer multiply and divide	10.7	9
Floating-point add and subtract	3.5	7
Floating-point multiply and divide	4.6	17
Logical operations	6.0	1
Branch instruction	20.0	1.5
Compare and shift	16.8	2

Compute the resulting MIPS rate for program X.

Exercise 4.3 There is a SPEC benchmark suite MPI2007 for evaluating the MPI performance of parallel systems for floating-point, compute-intensive programs. Visit the SPEC web page at `www.spec.org` and collect information on the benchmark programs included in the benchmark suite. Write a short summary for each of the benchmarks with computations performed, programming language used, MPI usage, and input description. What criteria were used to select the benchmarks? Which information is obtained by running the benchmarks?

Exercise 4.4 There is a SPEC benchmark suite to evaluate the performance of parallel systems with a shared address space based on OpenMP applications. Visit the SPEC web page at `www.spec.org` and collect information about this benchmark suite. Which applications are included and what information is obtained by running the benchmark?

Exercise 4.5 The SPEC CPU2006 is the standard benchmark suite to evaluate the performance of computer systems. Visit the SPEC web page at `www.spec.org` and collect the following information:

(a) Which benchmark programs are used in CINT2006 to evaluate the integer performance? Give a short characteristic of each of the benchmarks.
(b) Which benchmark programs are used in CFP2006 to evaluate the floating-point performance? Give a short characteristic of each of the benchmarks.
(c) Which performance results have been submitted for your favorite desktop computer?

Exercise 4.6 Consider a ring topology and assume that each processor can transmit at most one message at any time along an incoming or outgoing link (one-port communication). Show that the running time for a single-broadcast, a scatter operation, or a multi-broadcast takes time $\Theta(p)$. Show that a total exchange needs time $\Theta(p^2)$.

Exercise 4.7 Give an algorithm for a scatter operation on a linear array which sends the message from the root node for more distant nodes first and determine the asymptotic running time.

Exercise 4.8 Given a two-dimensional mesh with wraparound arrows forming a torus consisting of $n \times n$ nodes. Construct spanning trees for a multi-broadcast operation according to the construction in Sect. 4.3.2.2, p. 174, and give a corresponding algorithm for the communication operation which takes time $(n^2 - 1)/4$ for n odd and $n^2/4$ for n even [19].

Exercise 4.9 Consider a d-dimensional mesh network with $\sqrt[d]{p}$ processors in each of the d dimensions. Show that a multi-broadcast operation requires at least $\lceil (p-1)/d \rceil$ steps to be implemented. Construct an algorithm for the implementation of a multi-broadcast that performs the operation with this number of steps.

Exercise 4.10 Consider the construction of a spanning tree in Sect. 4.3.2, p. 173, and Fig. 4.4. Use this construction to determine the spanning tree for a five-dimensional hypercube network.

Exercise 4.11 For the construction of the spanning trees for the realization of a multi-broadcast operation on a d-dimensional hypercube network, we have used the relation

$$\binom{d}{k-1} - d \geq d$$

for $2 < k < d$ and $d \geq 5$, see Sect. 4.3.2, p. 180. Show by induction that this relation is true.
$\left(Hint: Itis \binom{d}{k-1} = \binom{d-1}{k-1} + \binom{d-1}{k-2} . \right)$

Exercise 4.12 Consider a complete binary tree with p processors [19].

a) Show that a single-broadcast operation takes time $\Theta(\log p)$.
b) Give an algorithm for a scatter operation with time $\Theta(p)$. (*Hint*: Send the more distant messages first.)
c) Show that an optimal algorithm for a multi-broadcast operation takes $p - 1$ time steps.

d) Show that a total exchange needs at least time $\Omega(p^2)$. (*Hint*: Count the number of messages that must be transmitted along the incoming links of a node.)
e) Show that a total exchange needs at most time $\Omega(p^2)$. (*Hint*: Use an embedding of a ring topology into the tree.)

Exercise 4.13 Consider a scalar product and a matrix–vector multiplication and derive the formula for the running time on a mesh topology.

Exercise 4.14 Develop a runtime function to capture the execution time of a parallel matrix–matrix computation $C = A \cdot B$ for a distributed address space. Assume a hypercube network as interconnection. Consider the following distributions for A and B:

(a) A is distributed in column-blockwise, B in row-blockwise order.
(b) Both A and B are distributed in checkerboard order.

Compare the resulting runtime functions and try to identify situations in which one or the other distribution results in a faster parallel program.

Exercise 4.15 The multi-prefix operation leads to the effect that each participating processor P_j obtains the value $\sigma + \sum_{i=1}^{j-1} \sigma_i$ where processor P_i contributes values σ_i and σ is the initial value of the memory location used, see also p. 188. Illustrate the effect of a multi-prefix operation with an exchange diagram similar to those used in Sect. 3.5.2. The effect of multi-prefix operations can be used for the implementation of parallel loops where each processor gets iterations to be executed. Explain this usage in more detail.

Chapter 5
Message-Passing Programming

The message-passing programming model is based on the abstraction of a parallel computer with a distributed address space where each processor has a local memory to which it has exclusive access, see Sect. 2.3.1. There is no global memory. Data exchange must be performed by message-passing: To transfer data from the local memory of one processor A to the local memory of another processor B, A must send a message containing the data to B, and B must receive the data in a buffer in its local memory. To guarantee portability of programs, no assumptions on the topology of the interconnection network is made. Instead, it is assumed that each processor can send a message to any other processor.

A message-passing program is executed by a set of processes where each process has its own local data. Usually, one process is executed on one processor or core of the execution platform. The number of processes is often fixed when starting the program. Each process can access its local data and can exchange information and data with other processes by sending and receiving messages. In principle, each of the processes could execute a different program (MPMD, *multiple program multiple data*). But to make program design easier, it is usually assumed that each of the processes executes the same program (SPMD, *single program, multiple data*), see also Sect. 2.2. In practice, this is not really a restriction, since each process can still execute different parts of the program, selected, for example, by its process rank.

The processes executing a message-passing program can exchange local data by using communication operations. These could be provided by a communication library. To activate a specific communication operation, the participating processes call the corresponding communication function provided by the library. In the simplest case, this could be a point-to-point transfer of data from a process A to a process B. In this case, A calls a send operation, and B calls a corresponding receive operation. Communication libraries often provide a large set of communication functions to support different point-to-point transfers and also global communication operations like broadcast in which more than two processes are involved, see Sect. 3.5.2 for a typical set of global communication operations.

A communication library could be vendor or hardware specific, but in most cases portable libraries are used, which define syntax and semantics of communication functions and which are supported for a large class of parallel computers. By far the

T. Rauber, G. Rünger, *Parallel Programming*,
DOI 10.1007/978-3-642-04818-0_5,

most popular portable communication library is MPI (*Message-Passing Interface*) [55, 56], but PVM (*Parallel Virtual Machine*) is also often used, see [63]. In this chapter, we give an introduction to MPI and show how parallel programs with MPI can be developed. The description includes point-to-point and global communication operations, but also more advanced features like process groups and communicators are covered.

5.1 Introduction to MPI

The Message-Passing Interface (MPI) is a standardization of a message-passing library interface specification. MPI defines the syntax and semantics of library routines for standard communication patterns as they have been considered in Sect. 3.5.2. Language bindings for C, C++, Fortran-77, and Fortran-95 are supported. In the following, we concentrate on the interface for C and describe the most important features. For a detailed description, we refer to the official MPI documents, see `www.mpi-forum.org`. There are two versions of the MPI standard: MPI-1 defines standard communication operations and is based on a static process model. MPI-2 extends MPI-1 and provides additional support for dynamic process management, one-sided communication, and parallel I/O. MPI is an interface specification for the syntax and semantics of communication operations, but leaves the details of the implementation open. Thus, different MPI libraries can use different implementations, possibly using specific optimizations for specific hardware platforms. For the programmer, MPI provides a standard interface, thus ensuring the portability of MPI programs. Freely available MPI libraries are MPICH (see `www-unix.mcs.anl.gov/mpi/mpich2`), LAM/MPI (see `www.lam-mpi.org`), and OpenMPI (see `www.open-mpi.org`).

In this section, we give an overview of MPI according to [55, 56]. An MPI program consists of a collection of processes that can exchange messages. For MPI-1, a static process model is used, which means that the number of processes is set when starting the MPI program and cannot be changed during program execution. Thus, MPI-1 does not support dynamic process creation during program execution. Such a feature is added by MPI-2. Normally, each processor of a parallel system executes one MPI process, and the number of MPI processes started should be adapted to the number of processors that are available. Typically, all MPI processes execute the same program in an SPMD style. In principle, each process can read and write data from/into files. For a coordinated I/O behavior, it is essential that only one specific process perform the input or output operations. To support portability, MPI programs should be written for an arbitrary number of processes. The actual number of processes used for a specific program execution is set when starting the program.

On many parallel systems, an MPI program can be started from the command line. The following two commands are common or widely used:

```
mpiexec -n 4 programname programarguments
mpirun -np 4 programname programarguments.
```

This call starts the MPI program `programname` with $p = 4$ processes. The specific command to start an MPI program on a parallel system can differ.

A significant part of the operations provided by MPI is the operations for the exchange of data between processes. In the following, we describe the most important MPI operations. For a more detailed description of all MPI operations, we refer to [135, 162, 163]. In particular the official description of the MPI standard provides many more details that cannot be covered in our short description, see [56]. Most examples given in this chapter are taken from these sources. Before describing the individual MPI operations, we first introduce some semantic terms that are used for the description of MPI operations:

- **Blocking operation**: An MPI communication operation is *blocking*, if return of control to the calling process indicates that all resources, such as buffers, specified in the call can be reused, e.g., for other operations. In particular, all state transitions initiated by a blocking operation are completed before control returns to the calling process.
- **Non-blocking operation**: An MPI communication operation is *non-blocking*, if the corresponding call may return before all effects of the operation are completed and before the resources used by the call can be reused. Thus, a call of a non-blocking operation only **starts** the operation. The operation itself is completed not before all state transitions caused are completed and the resources specified can be reused.

The terms *blocking* and *non-blocking* describe the behavior of operations from the *local* view of the executing process, without taking the effects on other processes into account. But it is also useful to consider the effect of communication operations from a *global* viewpoint. In this context, it is reasonable to distinguish between *synchronous* and *asynchronous* communications:

- **Synchronous communication**: The communication between a sending process and a receiving process is performed such that the communication operation does not complete before both processes have started their communication operation. This means in particular that the completion of a synchronous send indicates not only that the send buffer can be reused, but also that the receiving process has started the execution of the corresponding receive operation.
- **Asynchronous communication**: Using asynchronous communication, the sender can execute its communication operation without any coordination with the receiving process.

In the next section, we consider single transfer operations provided by MPI, which are also called point-to-point communication operations.

5.1.1 MPI Point-to-Point Communication

In MPI, all communication operations are executed using a **communicator**. A communicator represents a communication domain which is essentially a set of

processes that exchange messages between each other. In this section, we assume that the MPI default communicator MPI_COMM_WORLD is used for the communication. This communicator captures all processes executing a parallel program. In Sect. 5.3, the grouping of processes and the corresponding communicators are considered in more detail.

The most basic form of data exchange between processes is provided by point-to-point communication. Two processes participate in this communication operation: A sending process executes a send operation and a receiving process executes a corresponding receive operation. The send operation is *blocking* and has the syntax:

```
int MPI_Send(void *smessage,
             int count,
             MPI_Datatype datatype,
             int dest,
             int tag,
             MPI_Comm comm).
```

The parameters have the following meaning:

- smessage specifies a send buffer which contains the data elements to be sent in successive order;
- count is the number of elements to be sent from the send buffer;
- datatype is the data type of each entry of the send buffer; all entries have the same data type;
- dest specifies the rank of the target process which should receive the data; each process of a communicator has a unique rank; the ranks are numbered from 0 to the number of processes minus one;
- tag is a message tag which can be used by the receiver to distinguish different messages from the same sender;
- comm specifies the communicator used for the communication.

The size of the message in bytes can be computed by multiplying the number count of entries with the number of bytes used for type datatype. The tag parameter should be an integer value between 0 and 32,767. Larger values can be permitted by specific MPI libraries.

To receive a message, a process executes the following operation:

```
int MPI_Recv(void *rmessage,
             int count,
             MPI_Datatype datatype,
             int source,
             int tag,
             MPI_Comm comm,
             MPI_Status *status).
```

This operation is also blocking. The parameters have the following meaning:

- `rmessage` specifies the receive buffer in which the message should be stored;
- `count` is the maximum number of elements that should be received;
- `datatype` is the data type of the elements to be received;
- `source` specifies the rank of the sending process which sends the message;
- `tag` is the message tag that the message to be received must have;
- `comm` is the communicator used for the communication;
- `status` specifies a data structure which contains information about a message after the completion of the receive operation.

The predefined MPI data types and the corresponding C data types are shown in Table 5.1. There is no corresponding C data type for `MPI_PACKED` and `MPI_BYTE`. The type `MPI_BYTE` represents a single byte value. The type `MPI_PACKED` is used by special MPI pack operations.

Table 5.1 Predefined data types for MPI

MPI Datentyp	C-Datentyp
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_LONG_LONG_INT	long long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_UNSIGNED_LONG_LONG	unsigned long long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_WCHAR	wide char
MPI_PACKED	special data type for packing
MPI_BYTE	single byte value

By using `source = MPI_ANY_SOURCE`, a process can receive a message from any arbitrary process. Similarly, by using `tag = MPI_ANY_TAG`, a process can receive a message with an arbitrary tag. In both cases, the `status` data structure contains the information, from which process the message received has been sent and which tag has been used by the sender. After completion of `MPI_Recv()`, `status` contains the following information:

- `status.MPI_SOURCE` specifies the rank of the sending process;
- `status.MPI_TAG` specifies the tag of the message received;
- `status.MPI_ERROR` contains an error code.

The `status` data structure also contains information about the length of the message received. This can be obtained by calling the MPI function

```
int MPI_Get_count (MPI_Status *status,
                   MPI_Datatype datatype,
                   int *count_ptr),
```

where `status` is a pointer to the data structure `status` returned by `MPI_Recv()`. The function returns the number of elements received in the variable pointed to by `count_ptr`.

Internally a message transfer in MPI is usually performed in three steps:

1. The data elements to be sent are copied from the send buffer `smessage` specified as parameter into a system buffer of the MPI runtime system. The message is assembled by adding a header with information on the sending process, the receiving process, the tag, and the communicator used.
2. The message is sent via the network from the sending process to the receiving process.
3. At the receiving side, the data entries of the message are copied from the system buffer into the receive buffer `rmessage` specified by `MPI_Recv()`.

Both `MPI_Send()` and `MPI_Recv()` are *blocking*, *asynchronous* operations. This means that an `MPI_Recv()` operation can also be started when the corresponding `MPI_Send()` operation has not yet been started. The process executing the `MPI_Recv()` operation is blocked until the specified receive buffer contains the data elements sent. Similarly, an `MPI_Send()` operation can also be started when the corresponding `MPI_Recv()` operation has not yet been started. The process executing the `MPI_Send()` operation is blocked until the specified send buffer can be reused. The exact behavior depends on the specific MPI library used. The following two behaviors can often be observed:

- If the message is sent directly from the send buffer specified without using an internal system buffer, then the `MPI_Send()` operation is blocked until the entire message has been copied into a receive buffer at the receiving side. In particular, this requires that the receiving process has started the corresponding `MPI_Recv()` operation.
- If the message is first copied into an internal system buffer of the runtime system, the sender can continue its operations as soon as the copy operation into the system buffer is completed. Thus, the corresponding `MPI_Recv()` operation does not need to be started. This has the advantage that the sender is not blocked for a long period of time. The drawback of this version is that the system buffer needs additional memory space and that the copying into the system buffer requires additional execution time.

Example Figure 5.1 shows a first MPI program in which the process with rank 0 uses `MPI_Send()` to send a message to the process with rank 1. This process uses `MPI_Recv()` to receive a message. The MPI program shown is executed by all participating processes, i.e., each process executes the same program. But different processes may execute different program parts, e.g., depending on the values of local variables. The program defines a variable `status` of type `MPI_Status`, which is

```
#include <stdio.h>
#include <string.h>
#include "mpi.h"

int main (int argc, char *argv[]) {
   int my_rank, p, tag=0;
   char msg [20];
   MPI_Status status;

   MPI_Init (&argc, &argv);
   MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);
   MPI_Comm_size (MPI_COMM_WORLD, &p);
   if (my_rank == 0) {
      strcpy (msg, "Hello ");
      MPI_Send (msg, strlen(msg)+1, MPI_CHAR, 1, tag, MPI_COMM_WORLD);
   }
   if (my_rank == 1)
      MPI_Recv (msg, 20, MPI_CHAR, 0, tag, MPI_COMM_WORLD, &status);
   MPI_Finalize ();
}
```

Fig. 5.1 A first MPI program: message passing from process 0 to process 1

used for the `MPI_Recv()` operation. Any MPI program must include `<mpi.h>`. The MPI function `MPI_Init()` must be called before any other MPI function to initialize the MPI runtime system. The call `MPI_Comm_rank(MPI_COMM_WORLD, &my_rank)` returns the rank of the calling process in the communicator specified, which is `MPI_COMM_WORLD` here. The rank is returned in the variable `my_rank`. The function `MPI_Comm_size(MPI_COMM_WORLD, &p)` returns the total number of processes in the specified communicator in variable `p`. In the example program, different processes execute different parts of the program depending on their rank stored in `my_rank`: Process 0 executes a string copy and an `MPI_Send()` operation; process 1 executes a corresponding `MPI_Recv()` operation. The `MPI_Send()` operation specifies in its fourth parameter that the receiving process has rank 1. The `MPI_Recv()` operation specifies in its fourth parameter that the sending process should have rank 0. The last operation in the example program is `MPI_Finalize()` which should be the last MPI operation in any MPI program.

□

An important property to be fulfilled by any MPI library is that messages are delivered in the order in which they have been sent. If a sender sends two messages one after another to the same receiver and both messages fit to the first `MPI_Recv()` called by the receiver, the MPI runtime system ensures that the first message sent will always be received first. But this order can be disturbed if more than two processes are involved. This can be illustrated with the following program fragment:

```
/* example to demonstrate the order of receive operations */
MPI_Comm_rank (comm, &my_rank);
if (my_rank == 0) {
   MPI_Send (sendbuf1, count, MPI_INT, 2, tag, comm);
   MPI_Send (sendbuf2, count, MPI_INT, 1, tag, comm);
}
else if (my_rank == 1) {
   MPI_Recv (recvbuf1, count, MPI_INT, 0, tag, comm, &status);
   MPI_Send (recvbuf1, count, MPI_INT, 2, tag, comm);
}
else if (my_rank == 2) {
   MPI_Recv (recvbuf1, count, MPI_INT, MPI_ANY_SOURCE, tag, comm,
      &status);
   MPI_Recv (recvbuf2, count, MPI_INT, MPI_ANY_SOURCE, tag, comm,
      &status);
}
```

Process 0 first sends a message to process 2 and then to process 1. Process 1 receives a message from process 0 and forwards it to process 2. Process 2 receives two messages in the order in which they arrive using `MPI_ANY_SOURCE`. In this scenario, it can be expected that process 2 first receives the message that has been sent by process 0 directly to process 2, since process 0 sends this message first and since the second message sent by process 0 has to be forwarded by process 1 before arriving at process 2. But this must not necessarily be the case, since the first message sent by process 0 might be delayed because of a collision in the network whereas the second message sent by process 0 might be delivered without delay. Therefore, it can happen that process 2 first receives the message of process 0 that has been forwarded by process 1. Thus, if more than two processes are involved, there is no guaranteed delivery order. In the example, the expected order of arrival can be ensured if process 2 specifies the expected sender in the `MPI_Recv()` operation instead of `MPI_ANY_SOURCE`.

5.1.2 Deadlocks with Point-to-Point Communications

Send and receive operations must be used with care, since **deadlocks** can occur in ill-constructed programs. This can be illustrated by the following example:

```
/* program fragment which always causes a deadlock */
MPI_Comm_rank (comm, &my_rank);
if (my_rank == 0) {
   MPI_Recv (recvbuf, count, MPI_INT, 1, tag, comm, &status);
   MPI_Send (sendbuf, count, MPI_INT, 1, tag, comm);
}
else if (my_rank == 1) {
   MPI_Recv (recvbuf, count, MPI_INT, 0, tag, comm, &status);
   MPI_Send (sendbuf, count, MPI_INT, 0, tag, comm);
}
```

Both processes 0 and 1 execute an MPI_Recv() operation before an MPI_Send() operation. This leads to a deadlock because of mutual waiting: For process 0, the MPI_Send() operation can be started not before the preceding MPI_Recv() operation has been completed. This is only possible when process 1 executes its MPI_Send() operation. But this cannot happen because process 1 also has to complete its preceding MPI_Recv() operation first which can happen only if process 0 executes its MPI_Send() operation. Thus, cyclic waiting occurs, and this program always leads to a deadlock.

The occurrence of a deadlock might also depend on the question whether the runtime system uses internal system buffers or not. This can be illustrated by the following example:

```
/* program fragment for which the occurrence of a deadlock
   depends on the implementation */
MPI_Comm_rank (comm, &my_rank);
if (my_rank == 0) {
   MPI_Send (sendbuf, count, MPI_INT, 1, tag, comm);
   MPI_Recv (recvbuf, count, MPI_INT, 1, tag, comm, &status);
}
else if (my_rank == 1) {
   MPI_Send (sendbuf, count, MPI_INT, 0, tag, comm);
   MPI_Recv (recvbuf, count, MPI_INT, 0, tag, comm, &status);
}
```

Message transmission is performed correctly here without deadlock, if the MPI runtime system uses system buffers. In this case, the messages sent by processes 0 and 1 are first copied from the specified send buffer sendbuf into a system buffer before the actual transmission. After this copy operation, the MPI_Send() operation is completed because the send buffers can be reused. Thus, both processes 0 and 1 can execute their MPI_Recv() operation and no deadlock occurs. But a deadlock occurs, if the runtime system does not use system buffers or if the system buffers used are too small. In this case, none of the two processes can complete its MPI_Send() operation, since the corresponding MPI_Recv() cannot be executed by the other process.

A **secure implementation** which does not cause deadlocks even if no system buffers are used is the following:

```
/* program fragment that does not cause a deadlock */
MPI_Comm_rank (comm, &myrank);
if (my_rank == 0) {
   MPI_Send (sendbuf, count, MPI_INT, 1, tag, comm);
   MPI_Recv (recvbuf, count, MPI_INT, 1, tag, comm, &status);
}
else if (my_rank == 1) {
   MPI_Recv (recvbuf, count, MPI_INT, 0, tag, comm, &status);
   MPI_Send (sendbuf, count, MPI_INT, 0, tag, comm);
}
```

An MPI program is called **secure** if the correctness of the program does not depend on assumptions about specific properties of the MPI runtime system, like the existence of system buffers or the size of system buffers. Thus, secure MPI programs work correctly even if no system buffers are used. If more than two processes exchange messages such that each process sends and receives a message, the program must exactly specify in which order the send and receive operations are to be executed to avoid deadlocks. As example, we consider a program with p processes where process i sends a message to process $(i + 1) \bmod p$ and receives a message from process $(i - 1) \bmod p$ for $0 \leq i \leq p - 1$. Thus, the messages are sent in a logical ring. A secure implementation can be obtained if processes with an even rank first execute their send and then their receive operation, whereas processes with an odd rank first execute their receive and then their send operation. This leads to a communication with two phases and to the following exchange scheme for four processes:

Phase	Process 0	Process 1	Process 2	Process 3
1	`MPI_Send()` to 1	`MPI_Recv()` from 0	`MPI_Send()` to 3	`MPI_Recv()` from 2
2	`MPI_Recv()` from 3	`MPI_Send()` to 2	`MPI_Recv()` from 1	`MPI_Send()` to 0

The described execution order leads to a secure implementation also for an odd number of processes. For three processes, the following exchange scheme results:

Phase	Process 0	Process 1	Process 2
1	`MPI_Send()` to 1	`MPI_Recv()` from 0	`MPI_Send()` to 0
2	`MPI_Recv()` from 2	`MPI_Send()` to 2	-wait-
3		-wait-	`MPI_Recv()` from 1

In this scheme, some communication operations like the `MPI_Send()` operation of process 2 can be delayed because the receiver calls the corresponding `MPI_Recv()` operation at a later time. But a deadlock cannot occur.

In many situations, processes both send and receive data. MPI provides the following operations to support this behavior:

```
int MPI_Sendrecv (void *sendbuf,
                  int sendcount,
                  MPI_Datatype sendtype,
                  int dest,
                  int sendtag,
                  void *recvbuf,
                  int recvcount,
                  MPI_Datatype recvtype,
                  int source,
```

```
                    int recvtag,
                    MPI_Comm comm,
                    MPI_Status *status).
```

This operation is blocking and combines a send and a receive operation in one call. The parameters have the following meaning:

- `sendbuf` specifies a send buffer in which the data elements to be sent are stored;
- `sendcount` is the number of elements to be sent;
- `sendtype` is the data type of the elements in the send buffer;
- `dest` is the rank of the target process to which the data elements are sent;
- `sendtag` is the tag for the message to be sent;
- `recvbuf` is the receive buffer for the message to be received;
- `recvcount` is the maximum number of elements to be received;
- `recvtype` is the data type of elements to be received;
- `source` is the rank of the process from which the message is expected;
- `recvtag` is the expected tag of the message to be received;
- `comm` is the communicator used for the communication;
- `status` specifies the data structure to store the information on the message received.

Using `MPI_Sendrecv()`, the programmer does not need to worry about the order of the send and receive operations. The MPI runtime system guarantees deadlock freedom, also for the case that no internal system buffers are used. The parameters `sendbuf` and `recvbuf`, specifying the send and receive buffers of the executing process, must be disjoint, non-overlapping memory locations. But the buffers may have different lengths, and the entries stored may even contain elements of different data types. There is a variant of `MPI_Sendrecv()` for which the send buffer and the receive buffer are identical. This operation is also blocking and has the following syntax:

```
int MPI_Sendrecv_replace (void *buffer,
                          int count,
                          MPI_Datatype type,
                          int dest,
                          int sendtag,
                          int source,
                          int recvtag,
                          MPI_Comm comm,
                          MPI_Status *status).
```

Here, `buffer` specifies the buffer that is used as both send and receive buffer. For this function, `count` is the number of elements to be sent and to be received; these elements now should have identical type `type`.

5.1.3 Non-blocking Operations and Communication Modes

The use of blocking communication operations can lead to waiting times in which the blocked process does not perform useful work. For example, a process executing a blocking send operation must wait until the send buffer has been copied into a system buffer or even until the message has completely arrived at the receiving process if no system buffers are used. Often, it is desirable to fill the waiting times with useful operations of the waiting process, e.g., by overlapping communications and computations. This can be achieved by using *non-blocking communication operations*.

A **non-blocking send operation** initiates the sending of a message and returns control to the sending process as soon as possible. Upon return, the send operation has been started, but the send buffer specified cannot be reused safely, i.e., the transfer into an internal system buffer may still be in progress. A separate completion operation is provided to test whether the send operation has been completed locally. A non-blocking send has the advantage that control is returned as fast as possible to the calling process which can then execute other useful operations. A non-blocking send is performed by calling the following MPI function:

```
int MPI_Isend (void *buffer,
               int count,
               MPI_Datatype type,
               int dest,
               int tag,
               MPI_Comm comm,
               MPI_Request *request).
```

The parameters have the same meaning as for `MPI_Send()`. There is an additional parameter of type `MPI_Request` which denotes an opaque object that can be used for the identification of a specific communication operation. This request object is also used by the MPI runtime system to report information on the status of the communication operation.

A **non-blocking receive operation** initiates the receiving of a message and returns control to the receiving process as soon as possible. Upon return, the receive operation has been started and the runtime system has been informed that the receive buffer specified is ready to receive data. But the return of the call does not indicate that the receive buffer already contains the data, i.e., the message to be received cannot be used yet. A non-blocking receive is provided by MPI using the function

```
int MPI_Irecv (void *buffer,
               int count,
               MPI_Datatype type,
               int source,
               int tag,
               MPI_Comm comm,
               MPI_Request *request)
```

where the parameters have the same meaning as for MPI_Recv(). Again, a request object is used for the identification of the operation. Before reusing a send or receive buffer specified in a non-blocking send or receive operation, the calling process must test the completion of the operation. The request objects returned are used for the identification of the communication operations to be tested for completion. The following MPI function can be used to test for the completion of a non-blocking communication operation:

```
int MPI_Test (MPI_Request *request,
              int *flag,
              MPI_Status *status).
```

The call returns flag = 1 (true), if the communication operation identified by request has been completed. Otherwise, flag = 0 (false) is returned. If request denotes a receive operation and flag = 1 is returned, the parameter status contains information on the message received as described for MPI_Recv(). The parameter status is undefined if the specified receive operation has not yet been completed. If request denotes a send operation, all entries of status except status.MPI_ERROR are undefined. The MPI function

```
int MPI_Wait (MPI_Request *request, MPI_Status *status)
```

can be used to wait for the completion of a non-blocking communication operation. When calling this function, the calling process is blocked until the operation identified by request has been completed. For a non-blocking send operation, the send buffer can be reused after MPI_Wait() returns. Similarly for a non-blocking receive, the receive buffer contains the message after MPI_Wait() returns.

MPI also ensures for non-blocking communication operations that messages are non-overtaking. Blocking and non-blocking operations can be mixed, i.e., data sent by MPI_Isend() can be received by MPI_Recv() and data sent by MPI_Send() can be received by MPI_Irecv().

Example As example for the use of non-blocking communication operations, we consider the collection of information from different processes such that each process gets all available information [135]. We consider p processes and assume that each process has computed the same number of floating-point values. These values should be communicated such that each process gets the values of all other processes. To reach this goal, $p-1$ steps are performed and the processes are logically arranged in a ring. In the first step, each process sends its local data to its successor process in the ring. In the following steps, each process forwards the data that it has received in the previous step from its predecessor to its successor. After $p-1$ steps, each process has received all the data.

The steps to be performed are illustrated in Fig. 5.2 for four processes. For the implementation, we assume that each process provides its local data in an array x and that the entire data is collected in an array y of size p times the size of x.

Figure 5.3 shows an implementation with blocking send and receive operations. The size of the local data blocks of each process is given by parameter

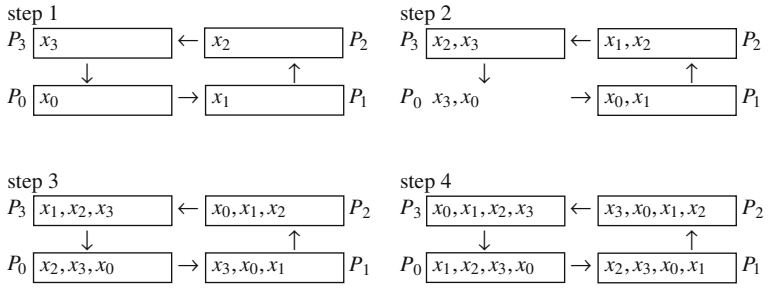


Fig. 5.2 Illustration for the collection of data in a logical ring structure for $p = 4$ processes

```
void Gather_ring (float x[], int blocksize, float y[])
{
   int i, p, my_rank, succ, pred;
   int send_offset, recv_offset;
   MPI_Status status;

   MPI_Comm_size (MPI_COMM_WORLD, &p);
   MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);
   for (i=0; i<blocksize; i++)
      y[i+my_rank * blocksize] = x[i];
   succ = (my_rank+1) % p;
   pred = (my_rank-1+p) % p;
   for (i=0; i<p-1; i++) {
      send_offset = ((my_rank-i+p) % p) * blocksize;
      recv_offset = ((my_rank-i-1+p) % p) * blocksize;
      MPI_Send (y+send_offset, blocksize, MPI_FLOAT, succ, 0,
                MPI_COMM_WORLD);
      MPI_Recv (y+recv_offset, blocksize, MPI_FLOAT, pred, 0,
                MPI_COMM_WORLD, &status);
   }
}
```

Fig. 5.3 MPI program for the collection of distributed data blocks. The participating processes are logically arranged as a ring. The communication is performed with *blocking* point-to-point operations. Deadlock freedom is ensured only if the MPI runtime system uses system buffers that are large enough

blocksize. First, each process copies its local block x into the corresponding position in y and determines its predecessor process pred as well as its successors process succ in the ring. Then, a loop with $p - 1$ steps is performed. In each step, the data block received in the previous step is sent to the successor process, and a new block is received from the predecessor process and stored in the next block position to the left in y. It should be noted that this implementation requires the use of system buffers that are large enough to store the data blocks to be sent.

An implementation with non-blocking communication operations is shown in Fig. 5.4. This implementation allows an overlapping of communication with local computations. In this example, the local computations overlapped are the computations of the positions of send_offset and recv_offset of the next blocks to be sent or to be received in array y. The send and receive operations are

```
void Gather_ring_nb (float x[], int blocksize, float y[])
{
   int i, p, my_rank, succ, pred;
   int send_offset, recv_offset;
   MPI_Status status;
   MPI_Request send_request, recv_request;

   MPI_Comm_size (MPI_COMM_WORLD, &p);
   MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);
   for (i=0; i<blocksize; i++)
      y[i+my_rank * blocksize] = x[i];
   succ = (my_rank+1) % p;
   pred = (my_rank-1+p) % p;
   send_offset = my_rank * blocksize;
   recv_offset = ((my_rank-1+p) % p) * blocksize;
   for (i=0; i<p-1; i++) {
      MPI_Isend (y+send_offset, blocksize, MPI_FLOAT, succ, 0,
         MPI_COMM_WORLD, &send_request);
      MPI_Irecv (y+recv_offset, blocksize, MPI_FLOAT, pred, 0,
         MPI_COMM_WORLD, &recv_request);
      send_offset = ((my_rank-i-1+p) % p) * blocksize;
      recv_offset = ((my_rank-i-2+p) % p) * blocksize;
      MPI_Wait (&send_request, &status);
      MPI_Wait (&recv_request, &status);
   }
}
```

Fig. 5.4 MPI program for the collection of distributed data blocks, see Fig. 5.3. Non-blocking communication operations are used instead of blocking operations

started with MPI_Isend() and MPI_Irecv(), respectively. After control returns from these operations, send_offset and recv_offset are re-computed and MPI_Wait() is used to wait for the completion of the send and receive operations. According to [135], the non-blocking version leads to a smaller execution time than the blocking version on an Intel Paragon and IBM SP2 machine. □

5.1.4 Communication Mode

MPI provides different **communication modes** for both blocking and non-blocking communication operations. These communication modes determine the coordination between a send and its corresponding receive operation. The following three modes are available.

5.1.4.1 Standard Mode

The communication operations described until now use the standard mode of communication. In this mode, the MPI runtime system decides whether outgoing messages are buffered in a local system buffer or not. The runtime system could, for example, decide to buffer small messages up to a predefined size, but not large messages. For the programmer, this means that he cannot rely on a buffering of messages. Hence, programs should be written in such a way that they also work if no buffering is used.

5.1.4.2 Synchronous Mode

In the standard mode, a send operation can be completed even if the corresponding receive operation has not yet been started (if system buffers are used). In contrast, in synchronous mode, a send operation will be completed not before the corresponding receive operation has been started and the receiving process has started to receive the data sent. Thus, the execution of a send and receive operation in synchronous mode leads to a form of synchronization between the sending and the receiving processes: The return of a send operation in synchronous mode indicates that the receiver has started to store the message in its local receive buffer. A blocking send operation in synchronous mode is provided in MPI by the function MPI_Ssend(), which has the same parameters as MPI_Send() with the same meaning. A non-blocking send operation in synchronous mode is provided by the MPI function MPI_Issend(), which has the same parameters as MPI_Isend() with the same meaning. Similar to a non-blocking send operation in standard mode, control is returned to the calling process as soon as possible, i.e., in synchronous mode there is **no synchronization** between MPI_Issend() and MPI_Irecv(). Instead, synchronization between sender and receiver is performed when the sender calls MPI_Wait(). When calling MPI_Wait() for a non-blocking send operation in synchronous mode, control is returned to the calling process not before the receiver has called the corresponding MPI_Recv() or MPI_Irecv() operation.

5.1.4.3 Buffered Mode

In buffered mode, the local execution and termination of a send operation is not influenced by non-local events as is the case for the synchronous mode and can be the case for standard mode if no or too small system buffers are used. Thus, when starting a send operation in buffered mode, control will be returned to the calling process even if the corresponding receive operation has not yet been started. Moreover, the send buffer can be reused immediately after control returns, even if a non-blocking send is used. If the corresponding receive operation has not yet been started, the runtime system must buffer the outgoing message. A blocking send operation in buffered mode is performed by calling the MPI function `MPI_Bsend()`, which has the same parameters as `MPI_Send()` with the same meaning. A non-blocking send operation in buffered mode is performed by calling `MPI_Ibsend()`, which has the same parameters as `MPI_Isend()`. In buffered mode, the buffer space to be used by the runtime system must be provided by the programmer. Thus, it is the programmer who is responsible that a sufficiently large buffer is available. In particular, a send operation in buffered mode may fail if the buffer provided by the programmer is too small to store the message. The buffer for the buffering of messages by the sender is provided by calling the MPI function

```
int MPI_Buffer_attach (void *buffer, int buffersize),
```

where `buffersize` is the size of the buffer `buffer` in bytes. Only one buffer can be attached by each process at a time. A buffer previously provided can be detached again by calling the function

```
int MPI_Buffer_detach (void *buffer, int *buffersize),
```

where `buffer` is the *address* of the buffer pointer used in `MPI_Buffer_attach()`; the size of the buffer detached is returned in the parameter `buffer-size`. A process calling `MPI_Buffer_detach()` is blocked until all messages that are currently stored in the buffer have been transmitted.
For receive operations, MPI provides the standard mode only.

5.2 Collective Communication Operations

A communication operation is called *collective* or *global* if all or a subset of the processes of a parallel program are involved. In Sect. 3.5.2, we have shown global communication operations which are often used. In this section, we show how these communication operations can be used in MPI. The following table gives an overview of the operations supported:

Global communication operation	MPI function
Broadcast operation	`MPI_Bcast()`
Accumulation operation	`MPI_Reduce()`
Gather operation	`MPI_Gather()`
Scatter operation	`MPI_Scatter()`
Multi-broadcast operation	`MPI_Allgather()`
Multi-accumulation operation	`MPI_Allreduce()`
Total exchange	`MPI_Alltoall()`

5.2.1 Collective Communication in MPI

5.2.1.1 Broadcast Operation

For a broadcast operation, one specific process of a group of processes sends the same data block to all other processes of the group, see Sect. 3.5.2. In MPI, a broadcast is performed by calling the following MPI function:

```
int MPI_Bcast (void *message,
               int count,
               MPI_Datatype type,
               int root,
               MPI_Comm comm),
```

where `root` denotes the process which sends the data block. This process provides the data block to be sent in parameter `message`. The other processes specify in `message` their receive buffer. The parameter `count` denotes the number of elements in the data block, `type` is the data type of the elements of the data block. `MPI_Bcast()` is a *collective* communication operation, i.e., each process of the communicator `comm` must call the `MPI_Bcast()` operation. Each process must specify the same `root` process and must use the same communicator. Similarly, the type `type` and number `count` specified by any process including the root process must be the same for all processes. Data blocks sent by `MPI_Bcast()` *cannot* be received by an `MPI_Recv()` operation.

As can be seen in the parameter list of `MPI_Bcast()`, no tag information is used as is the case for point-to-point communication operations. Thus, the receiving processes cannot distinguish between different broadcast messages based on tags.

The MPI runtime system guarantees that broadcast messages are received in the same order in which they have been sent by the root process, even if the corresponding broadcast operations are not executed at the same time. Figure 5.5 shows as example a program part in which process 0 sends two data blocks `x` and `y` by two successive broadcast operations to process 1 and process 2 [135].

Process 1 first performs local computations by `local_work()` and then stores the first broadcast message in its local variable `y`, the second one in `x`. Process 2 stores the broadcast messages in the same local variables from which they have been sent by process 0. Thus, process 1 will store the messages in other local variables as process 2. Although there is no explicit synchronization between the processes

Fig. 5.5 Example for the receive order with several broadcast operations

```
if (my_rank == 0) {
   MPI_Bcast (&x, 1, MPI_INT, 0, comm);
   MPI_Bcast (&y, 1, MPI_INT, 0, comm);
   local_work ();
}
else if (my_rank == 1) {
   local_work ();
   MPI_Bcast (&y, 1, MPI_INT, 0, comm);
   MPI_Bcast (&x, 1, MPI_INT, 0, comm);
}
else if (my_rank == 2) {
   local_work ();
   MPI_Bcast (&x, 1, MPI_INT, 0, comm);
   MPI_Bcast (&y, 1, MPI_INT, 0, comm);
}
```

executing `MPI_Bcast()`, synchronous execution semantics is used, i.e., the order of the `MPI_Bcast()` operations is such as if there were a synchronization between the executing processes.

Collective MPI communication operations are always *blocking*; no non-blocking versions are provided as is the case for point-to-point operations. The main reason for this is to avoid a large number of additional MPI functions. For the same reason, only the standard modus is supported for collective communication operations. A process participating in a collective communication operation can complete the operation and return control as soon as its local participation has been completed, no matter what the status of the other participating processes is. For the root process, this means that control can be returned as soon as the message has been copied into a system buffer and the send buffer specified as parameter can be reused. The other processes need not have received the message before the root process can continue its computations. For a receiving process, this means that control can be returned as soon as the message has been transferred into the local receive buffer, even if other receiving processes have not even started their corresponding `MPI_Bcast()` operation. Thus, the execution of a collective communication operation does not involve a synchronization of the participating processes.

5.2.1.2 Reduction Operation

An *accumulation* operation is also called *global reduction* operation. For such an operation, each participating process provides a block of data that is combined with the other blocks using a binary reduction operation. The accumulated result is collected at a root process, see also Sect. 3.5.2. In MPI, a global reduction operation is performed by letting each participating process call the function

```
int MPI_Reduce (void *sendbuf,
                void *recvbuf,
                int count,
                MPI_Datatype type,
```

```
                MPI_Op op,
                int root,
                MPI_Comm comm),
```

where `sendbuf` is a send buffer in which each process provides its local data for the reduction. The parameter `recvbuf` specifies the receive buffer which is provided by the root process `root`. The parameter `count` specifies the number of elements provided by each process; `type` is the data type of each of these elements. The parameter `op` specifies the reduction operation to be performed for the accumulation. This must be an *associative* operation. MPI provides a number of predefined reduction operations which are also *commutative*:

Representation	Operation
`MPI_MAX`	Maximum
`MPI_MIN`	Minimum
`MPI_SUM`	Sum
`MPI_PROD`	Product
`MPI_LAND`	Logical and
`MPI_BAND`	Bit-wise and
`MPI_LOR`	Logical or
`MPI_BOR`	Bit-wise or
`MPI_LXOR`	Logical exclusive or
`MPI_BXOR`	Bit-wise exclusive or
`MPI_MAXLOC`	Maximum value and corresponding index
`MPI_MINLOC`	Minimum value and corresponding index

The predefined reduction operations `MPI_MAXLOC` and `MPI_MINLOC` can be used to determine a global maximum or minimum value and also an additional index attached to this value. This will be used in Chap. 7 in Gaussian elimination to determine a global pivot element of a row as well as the process which owns this pivot element and which is then used as the root of a broadcast operation. In this case, the additional index value is a process rank. Another use could be to determine the maximum value of a distributed array as well as the corresponding index position. In this case, the additional index value is an array index. The operation defined by `MPI_MAXLOC` is

$$(u, i) \circ_{\max} (v, j) = (w, k),$$
$$\text{where } w = \max(u, v) \text{ and } k = \begin{cases} i & \text{if } u > v \\ \min(i, j) & \text{if } u = v \\ j & \text{if } u < v \end{cases}.$$

Analogously, the operation defined by `MPI_MINLOC` is

$$(u, i) \circ_{\min} (v, j) = (w, k),$$
$$\text{where } w = \min(u, v) \text{ and } k = \begin{cases} i & \text{if } u < v \\ \min(i, j) & \text{if } u = v \\ j & \text{if } u > v \end{cases}.$$

Thus, both operations work on pairs of values, consisting of a value and an index. Therefore the data type provided as parameter of `MPI_Reduce()` must represent such a pair of values. MPI provides the following pairs of data types:

`MPI_FLOAT_INT`	`(float,int)`
`MPI_DOUBLE_INT`	`(double,int)`
`MPI_LONG_INT`	`(long,int)`
`MPI_SHORT_INT`	`(short,int)`
`MPI_LONG_DOUBLE_INT`	`(long double,int)`
`MPI_2INT`	`(int,int)`

For an `MPI_Reduce()` operation, all participating processes must specify the same values for the parameters `count`, `type`, `op`, and `root`. The send buffers `sendbuf` and the receive buffer `recvbuf` must have the same size. At the root process, they must denote disjoint memory areas. An in-place version can be activated by passing `MPI_IN_PLACE` for `sendbuf` at the root process. In this case, the input data block is taken from the `recvbuf` parameter at the root process, and the resulting accumulated value then replaces this input data block after the completion of `MPI_Reduce()`.

Example As example, we consider the use of a global reduction operation using `MPI_MAXLOC`, see Fig. 5.6. Each process has an array of 30 values of type `double`, stored in array `ain` of length 30. The program part computes the maximum value for each of the 30 array positions as well as the rank of the process that stores this

```
double ain[30], aout[30];
int ind[30];
struct {double val; int rank;} in[30], out[30];
int i, my_rank, root=0;

MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);
for (i=0; i<30; i++) {
   in[i].val = ain[i];
   in[i].rank = my_rank;
}
MPI_Reduce(in,out,30,MPI_DOUBLE_INT,MPI_MAXLOC,root,MPI_COMM_WORLD);
if (my_rank == root)
   for (i=0; i<30; i++) {
      aout[i] = out[i].val;
      ind[i] = out[i].rank;
   }
```

Fig. 5.6 Example for the use of `MPI_Reduce()` using `MPI_MAXLOC` as reduction operator

maximum value. The information is collected at process 0: The maximum values are stored in array aout and the corresponding process ranks are stored in array ind. For the collection of the information based on value pairs, a data structure is defined for the elements of arrays in and out, consisting of a double and an int value. □

MPI supports the definition of user-defined reduction operations using the following MPI function:

```
int MPI_Op_create (MPI_User_function *function,
                   int commute,
                   MPI_Op *op).
```

The parameter function specifies a user-defined function which must define the following four parameters:

```
void *in, void *out, int *len, MPI_Datatype *type.
```

The user-defined function must be associative. The parameter commute specifies whether the function is also commutative (commute=1) or not (commute=0). The call of MPI_Op_create() returns a reduction operation op which can then be used as parameter of MPI_Reduce().

Example We consider the parallel computation of the scalar product of two vectors x and y of length m using p processes. Both vectors are partitioned into blocks of size local_m = m/p. Each block is stored by a separate process such that each process stores its local blocks of x and y in local vectors local_x and local_y. Thus, the process with rank my_rank stores the following parts of x and y:

```
local_x[j] = x[j + my_rank * local_m];
local_y[j] = y[j + my_rank * local_m];
```

for $0 \leq j <$ local_m.

```
int j, m, p, local_m;
float local_dot, dot;
float local_x[100], local_y[100];
MPI_Status status;

MPI_Comm_rank( MPI_COMM_WORLD, &my_rank);
MPI_Comm_size( MPI_COMM_WORLD, &p);
if (my_rank == 0) scanf("%d",&m);
local_m = m/p;
local_dot = 0.0;
for (j=0; j < local_m; j++)
    local_dot = local_dot + local_x[j] * local_y[j];
MPI_Reduce(&local_dot, &dot,1, MPI_FLOAT, MPI_SUM,0, MPI_COMM_WORLD);
```

Fig. 5.7 MPI program for the parallel computation of a scalar product

Figure 5.7 shows a program part for the computation of a scalar product. Each process executes this program part and computes a scalar product for its local blocks in `local_x` and `local_y`. The result is stored in `local_dot`. An `MPI_Reduce()` operation with reduction operation `MPI_SUM` is then used to add up the local results. The final result is collected at process 0 in variable `dot`. □

5.2.1.3 Gather Operation

For a gather operation, each process provides a block of data collected at a root process, see Sect. 3.5.2. In contrast to `MPI_Reduce()`, no reduction operation is applied. Thus, for p processes, the data block collected at the root process is p times larger than the individual blocks provided by each process. A gather operation is performed by calling the following MPI function :

```
int MPI_Gather(void *sendbuf,
               int sendcount,
               MPI_Datatype sendtype,
               void *recvbuf,
               int recvcount,
               MPI_Datatype recvtype,
               int root,
               MPI_Comm comm).
```

The parameter `sendbuf` specifies the send buffer which is provided by each participating process. Each process provides `sendcount` elements of type `sendtype`. The parameter `recvbuf` is the receive buffer that is provided by the root process. No other process must provide a receive buffer. The root process receives `recvcount` elements of type `recvtype` from each process of communicator `comm` and stores them in the order of the ranks of the processes according to `comm`. For p processes the effect of the `MPI_Gather()` call can also be achieved if each process, including the root process, calls a send operation

```
MPI_Send (sendbuf, sendcount, sendtype, root, my_rank, comm)
```

and the root process executes p receive operations

```
MPI_Recv (recvbuf+i*recvcount*extent,
          recvcount, recvtype, i, i, comm, &status),
```

where `i` enumerates all processes of `comm`. The number of bytes used for each element of the data blocks is stored in `extend` and can be determined by calling the function `MPI_Type_extent(recvtype, &extent)`. For a correct execution of `MPI_Gather()`, each process must specify the same root process `root`. Moreover, each process must specify the same element data type and the same number of elements to be sent. Figure 5.8 shows a program part in which process 0 collects 100 integer values from each process of a communicator.

```
MPI_Comm comm;
int sendbuf[100], my_rank, root = 0, gsize, *rbuf;
MPI_Comm_rank (comm, &my_rank);
if (my_rank == root) {
   MPI_Comm_size (comm, &gsize);
   rbuf = (int *) malloc (gsize*100*sizeof(int));
}
MPI_Gather(sendbuf, 100, MPI_INT, rbuf, 100, MPI_INT, root, comm);
```

Fig. 5.8 Example for the application of `MPI_Gather()`

MPI provides a variant of `MPI_Gather()` for which each process can provide a *different* number of elements to be collected. The variant is `MPI_Gatherv()`, which uses the same parameters as `MPI_Gather()` with the following two changes:

- the integer parameter `recvcount` is replaced by an integer array `recvcounts` of length p where `recvcounts[i]` denotes the number of elements provided by process `i`;
- there is an additional parameter `displs` after `recvcounts`. This is also an integer array of length p and `displs[i]` specifies at which position of the receive buffer of the root process the data block of process `i` is stored. Only the root process must specify the array parameters `recvcounts` and `displs`.

The effect of an `MPI_Gatherv()` operation can also be achieved if each process executes the send operation described above and the root process executes the following p receive operations:

```
MPI_Recv(recvbuf+displs[i]*extent, recvcounts[i], recvtype, i, i,
         comm, &status).
```

For a correct execution of `MPI_Gatherv()`, the parameter `sendcount` specified by process `i` must be equal to the value of `recvcounts[i]` specified by the root process. Moreover, the send and receive types must be identical for all processes. The array parameters `recvcounts` and `displs` specified by the root process must be chosen such that no location in the receive buffer is written more than once, i.e., an overlapping of received data blocks is not allowed.

Figure 5.9 shows an example for the use of `MPI_Gatherv()` which is a generalization of the example in Fig. 5.8: Each process provides 100 integer values, but the blocks received are stored in the receive buffer in such a way that there is a free gap between neighboring blocks; the size of the gaps can be controlled by parameter `displs`. In Fig. 5.9, `stride` is used to define the size of the gap, and the gap size is set to 10. An error occurs for `stride` $<$ `100`, since this would lead to an overlapping in the receive buffer.

```
MPI_Comm comm;
int sbuf[100];
int my_rank, root = 0, gsize, *rbuf, *displs, *rcounts, stride=110;
MPI_Comm_rank (comm, &my_rank);
if  (my_rank == root) {
    MPI_Comm_size (comm, &gsize);
    rbuf = (int *) malloc(gsize*stride*sizeof(int));
    displs = (int *) malloc(gsize*sizeof(int));
    rcounts = (int *) malloc(gsize*sizeof(int));
    for (i = 0; i < gsize; i++) {
        displs[i] = i*stride;
        rcounts[i] = 100;
    }
}
MPI_Gatherv(sbuf,100,MPI_INT,rbuf,rcounts,displs,MPI_INT,root,comm);
```

Fig. 5.9 Example for the use of `MPI_Gatherv()`

5.2.1.4 Scatter Operation

For a scatter operation, a root process provides a different data block for each participating process. By executing the scatter operation, the data blocks are distributed to these processes, see Sect. 3.5.2. In MPI, a scatter operation is performed by calling

```
int MPI_Scatter (void *sendbuf,
                 int sendcount,
                 MPI_Datatype sendtype,
                 void *recvbuf,
                 int recvcount,
                 MPI_Datatype recvtype,
                 int root,
                 MPI_Comm comm),
```

where `sendbuf` is the send buffer provided by the root process `root` which contains a data block for each process of the communicator `comm`. Each data block contains `sendcount` elements of type `sendtype`. In the send buffer, the blocks are ordered in rank order of the receiving process. The data blocks are received in the receive buffer `recvbuf` provided by the corresponding process. Each participating process including the root process must provide such a receive buffer. For p processes, the effects of `MPI_Scatter()` can also be achieved by letting the root process execute p send operations

```
MPI_Send (sendbuf+i*sendcount*extent, sendcount, sendtype, i, i,
          comm)
```

for `i` $= 0, \ldots, p-1$. Each participating process executes the corresponding receive operation

```
MPI_Recv (recvbuf, recvcount, recvtype, root, my_rank, comm,
          &status).
```

For a correct execution of MPI_Scatter(), each process must specify the same root, the same data types, and the same number of elements.

Similar to MPI_Gather(), there is a generalized version MPI_Scatterv() of MPI_Scatter() for which the root process can provide data blocks of different sizes. MPI_Scatterv() uses the same parameters as MPI_Scatter() with the following two changes:

- The integer parameter sendcount is replaced by the integer array sendcounts where sendcounts[i] denotes the number of elements sent to process i for $i = 0, \ldots, p-1$.
- There is an additional parameter displs after sendcounts which is also an integer array with p entries; displs[i] specifies from which position in the send buffer of the root process the data block for process i should be taken.

The effect of an MPI_Scatterv() operation can also be achieved by point-to-point operations: The root process executes p send operations

```
MPI_Send (sendbuf+displs[i]*extent,sendcounts[i],sendtype,i,
          i,comm)
```

and each process executes the receive operation described above.

For a correct execution of MPI_Scatterv(), the entry sendcounts[i] specified by the root process for process i must be equal to the value of recvcount specified by process i. In accordance with MPI_Gatherv(), it is required that the arrays sendcounts and displs are chosen such that no entry of the send buffer is sent to more than one process. This restriction is imposed for symmetry reasons with MPI_Gatherv() although this is not essential for a correct behavior. The program in Fig. 5.10 illustrates the use of a scatter operation. Process 0 distributes

```
MPI_Comm comm;
int rbuf[100];
int my_rank, root = 0, gsize, *sbuf, *displs, *scounts, stride=110;
MPI_Comm_rank (comm, &my_rank);
if (my_rank == root) {
   MPI_Comm_size (comm, &gsize);
   sbuf = (int *) malloc(gsize*stride*sizeof(int));
   displs = (int *) malloc(gsize*sizeof(int));
   scounts = (int *) malloc(gsize*sizeof(int));
   for (i=0; i<gsize; i++) {
       displs[i] = i*stride; scounts[i]=100;
   }
}
MPI_Scatterv(sbuf,scounts,displs,MPI_INT,rbuf,100,MPI_INT,root,comm);
```

Fig. 5.10 Example for the use of an MPI_Scatterv() operation

100 integer values to each other process such that there is a gap of 10 elements between neighboring send blocks.

5.2.1.5 Multi-broadcast Operation

For a multi-broadcast operation, each participating process contributes a block of data which could, for example, be a partial result from a local computation. By executing the multi-broadcast operation, all blocks will be provided to all processes. There is no distinguished root process, since each process obtains all blocks provided. In MPI, a multi-broadcast operation is performed by calling the function

```
int MPI_Allgather (void *sendbuf,
                   int sendcount,
                   MPI_Datatype sendtype,
                   void *recvbuf,
                   int recvcount,
                   MPI_Datatype recvtype,
                   MPI_Comm comm),
```

where `sendbuf` is the send buffer provided by each process containing the block of data. The send buffer contains `sendcount` elements of type `sendtype`. Each process also provides a receive buffer `recvbuf` in which all received data blocks are collected in the order of the ranks of the sending processes. The values of the parameters `sendcount` and `sendtype` must be the same as the values of `recvcount` and `recvtype`. In the following example, each process contributes a send buffer with 100 integer values which are collected by a multi-broadcast operation at each process:

```
int sbuf[100], gsize, *rbuf;
MPI_Comm_size (comm, &gsize);
rbuf = (int*) malloc (gsize*100*sizeof(int));
MPI_Allgather (sbuf, 100, MPI_INT, rbuf, 100, MPI_INT, comm);
```

For an `MPI_Allgather()` operation, each process must contribute a data block of the same size. There is a vector version of `MPI_Allgather()` which allows each process to contribute a data block of a different size. This vector version is obtained by a similar generalization as `MPI_Gatherv()` and is performed by calling the following function:

```
int MPI_Allgatherv (void *sendbuf,
                    int sendcount,
                    MPI_Datatype sendtype,
                    void *recvbuf,
                    int *recvcounts,
                    int *displs,
                    MPI_Datatype recvtype,
                    MPI_Comm comm).
```

The parameters have the same meaning as for `MPI_Gatherv()`.

5.2.1.6 Multi-accumulation Operation

For a multi-accumulation operation, each participating process performs a separate single-accumulation operation for which each process provides a different block of data, see Sect. 3.5.2. MPI provides a version of multi-accumulation with a restricted functionality: Each process provides the same data block for each single-accumulation operation. This can be illustrated by the following diagram:

$$\begin{array}{ll} P_0 : x_0 & P_0 : x_0 + x_1 + \cdots + x_{p-1} \\ P_1 : x_1 & P_1 : x_0 + x_1 + \cdots + x_{p-1} \\ \vdots \quad \overset{MPI-accumulation(+)}{\Longrightarrow} & \vdots \\ P_{p-1} : x_n & P_{p-1} : x_0 + x_1 + \cdots + x_{p-1} \end{array}$$

In contrast to the general version described in Sect. 3.5.2, each of the processes $P_0, \ldots, P_{p-1}$ only provides one data block for $k = 0, \ldots, p-1$, expressed as $P_k : x_k$. After the operation, each process has accumulated the *same* result block, represented by $P_k : x_0 + x_1 + \cdots + x_{p-1}$. Thus, a multi-accumulation operation in MPI has the same effect as a single-accumulation operation followed by a single-broadcast operation which distributes the accumulated data block to all processes. The MPI operation provided has the following syntax:

```
int MPI_Allreduce (void *sendbuf,
                   void *recvbuf,
                   int count,
                   MPI_Datatype type,
                   MPI_Op op,
                   MPI_Comm comm),
```

where `sendbuf` is the send buffer in which each process provides its local data block. The parameter `recvbuf` specifies the receive buffer in which each process of the communicator `comm` collects the accumulated result. Both buffers contain `count` elements of type `type`. The reduction operation `op` is used. Each process must specify the same size and type for the data block.

Example We consider the use of a multi-accumulation operation for the parallel computation of a matrix–vector multiplication $c = A \cdot b$ of an $n \times m$ matrix A with an m-dimensional vector b. The result is stored in the n-dimensional vector c. We assume that A is distributed in a column-oriented blockwise way such that each of the p processes stores `local_m = m/p` contiguous columns of A in its local memory, see also Sect. 3.4 on data distributions. Correspondingly, vector b is distributed in a blockwise way among the processes. The matrix–vector multiplication is performed in parallel as described in Sect. 3.6, see also Fig. 3.13. Figure 5.11 shows an outline of an MPI implementation. The blocks of columns stored by each process are stored in the two-dimensional array `a` which contains `n` rows and `local_m` columns. Each process stores its local columns consecutively in this array. The one-dimensional array `local_b` contains for each process its block

Fig. 5.11 MPI program piece to compute a matrix–vector multiplication with a column-blockwise distribution of the matrix using an `MPI_Allreduce()` operation

```
int m, local_m, n, p;
float a[MAX_N][MAX_LOC_M], local_b[MAX_LOC_M];
float c[MAX_N], sum[MAX_N];
local_m = m/p;
for (i=0; i<n; i++) {
   sum[i] = 0;
   for (j=0; j<local_m; j++)
      sum[i] = sum[i] + a[i][j]*local_b[j];
}
MPI_Allreduce (sum, c, n, MPI_FLOAT, MPI_SUM, comm);
```

of b of length `local_m`. Each process computes n partial scalar products for its local block of columns using partial vectors of length `local_m`. The global accumulation to the final result is performed with an `MPI_Allreduce()` operation, providing the result to all processes in a replicated way. □

5.2.1.7 Total Exchange

For a total exchange operation, each process provides a different block of data for each other process, see Sect. 3.5.2. The operation has the same effect as if each process performs a separate scatter operation (sender view) or as if each process performs a separate gather operation (receiver view). In MPI, a total exchange is performed by calling the function

```
int MPI_Alltoall (void *sendbuf,
                  int sendcount,
                  MPI_Datatype sendtype,
                  void *recvbuf,
                  int recvcount,
                  MPI_Datatype recvtype,
                  MPI_Comm comm),
```

where `sendbuf` is the send buffer in which each process provides for each process (including itself) a block of data with `sendcount` elements of type `sendtype`. The blocks are arranged in rank order of the target process. Each process also provides a receive buffer `recvbuf` in which the data blocks received from the other processes are stored. Again, the blocks received are stored in rank order of the sending processes. For p processes, the effect of a total exchange can also be achieved if each of the p processes executes p send operations

```
MPI_Send (sendbuf+i*sendcount*extent, sendcount, sendtype,
          i, my_rank, comm)
```

as well as p receive operations

```
MPI_Recv (recvbuf+i*recvcount*extent, recvcount, recvtype,
          i, i, comm, &status),
```

where i is the rank of one of the p processes and therefore lies between 0 and $p-1$.

For a correct execution, each participating process must provide for each other process data blocks of the same size and must also receive from each other process data blocks of the same size. Thus, all processes must specify the same values for sendcount and recvcount. Similarly, sendtype and recvtype must be the same for all processes. If data blocks of different sizes should be exchanged, the vector version must be used. This has the following syntax:

```
int MPI_Alltoallv (void *sendbuf,
                   int *scounts,
                   int *sdispls,
                   MPI_Datatype sendtype,
                   void *recvbuf,
                   int *rcounts,
                   int *rdispls,
                   MPI_Datatype recvtype,
                   MPI_Comm comm).
```

For each process i, the entry scounts[j] specifies how many elements of type sendtype process i sends to process j. The entry sdispls[j] specifies the start position of the data block for process j in the send buffer of process i. The entry rcounts[j] at process i specifies how many elements of type recvtype process i receives from process j. The entry rdispls[j] at process i specifies at which position in the receive buffer of process i the data block from process j is stored.

For a correct execution of MPI_Alltoallv(), scounts[j] at process i must have the same value as rcounts[i] at process j. For p processes, the effect of Alltoallv() can also be achieved, if each of the processes executes p send operations

```
MPI_Send (sendbuf+sdispls[i]*sextent, scounts[i],
          sendtype, i, my_rank, comm)
```

and p receive operations

```
MPI_Recv (recvbuf+rdispls[i]*rextent, rcounts[i],
          recvtype, i, i, comm, &status),
```

where i is the rank of one of the p processes and therefore lies between 0 and $p-1$.

5.2.2 Deadlocks with Collective Communication

Similar to single transfer operations, different behavior can be observed for collective communication operations, depending on the use of internal system buffers by the MPI implementation. A careless use of collective communication operations may lead to **deadlocks**, see also Sect. 3.7.4 (p. 140) for the occurrence of deadlocks with single transfer operations. This can be illustrated for `MPI_Bcast()` operations: We consider two MPI processes which execute two `MPI_Bcast()` operations in opposite order

```
switch (my_rank) {
case 0: MPI_Bcast (buf1, count, type, 0, comm);
        MPI_Bcast (buf2, count, type, 1, comm);
        break;
case 1: MPI_Bcast (buf2, count, type, 1, comm);
        MPI_Bcast (buf1, count, type, 0, comm);
}
```

Executing this piece of program may lead to two different error situations:

1. The MPI runtime system may match the first `MPI_Bcast()` call of each process. Doing this results in an error, since the two processes specify different roots.
2. The runtime system may match the `MPI_Bcast()` calls with the same root, as it has probably been intended by the programmer. Then a **deadlock** may occur if no system buffers are used or if the system buffers are too small. Collective communication operations are always **blocking**; thus, the operations are *synchronizing* if no or too small system buffers are used. Therefore, the first call of `MPI_Bcast()` blocks the process with rank 0 until the process with rank 1 has called the corresponding `MPI_Bcast()` with the same root. But this cannot happen, since process 1 is blocked due to its first `MPI_Bcast()` operation, waiting for process 0 to call its second `MPI_Bcast()`. Thus, a classical deadlock situation with cyclic waiting results.

The error or deadlock situation can be avoided in this example by letting the participating processes call the matching collective communication operations in the same order.

Deadlocks can also occur when mixing collective communication and single-transfer operations. This can be illustrated by the following example:

```
switch (my_rank) {
case 0: MPI_Bcast (buf1, count, type, 0, comm);
        MPI_Send (buf2, count, type, 1, tag, comm);
        break;
case 1: MPI_Recv (buf2, count, type, 0, tag, comm, &status);
        MPI_Bcast (buf1, count, type, 0, comm);
}
```

If no system buffers are used by the MPI implementation, a deadlock because of cyclic waiting occurs: Process 0 blocks when executing `MPI_Bcast()`, until process 1 executes the corresponding `MPI_Bcast()` operation. Process 1 blocks when executing `MPI_Recv()` until process 0 executes the corresponding `MPI_Send()` operation, resulting in cyclic waiting. This can be avoided if both processes execute their corresponding communication operations in the same order.

The **synchronization behavior** of collective communication operations depends on the use of system buffers by the MPI runtime system. If no internal system buffers are used or if the system buffers are too small, collective communication operations may lead to the synchronization of the participating processes. If system buffers are used, there is not necessarily a synchronization. This can be illustrated by the following example:

```
switch (my_rank) {
case 0: MPI_Bcast (buf1, count, type, 0, comm);
        MPI_Send (buf2, count, type, 1, tag, comm);
        break;
case 1: MPI_Recv (buf2, count, type, MPI_ANY_SOURCE, tag,
           comm, &status);
        MPI_Bcast (buf1, count, type, 0, comm);
        MPI_Recv (buf2, count, type, MPI_ANY_SOURCE, tag,
           comm, &status);
        break;
case 2: MPI_Send (buf2, count, type, 1, tag, comm);
        MPI_Bcast (buf1, count, type, 0, comm);
}
```

After having executed `MPI_Bcast()`, process 0 sends a message to process 1 using `MPI_Send()`. Process 2 sends a message to process 1 before executing an `MPI_Bcast()` operation. Process 1 receives two messages from `MPI_ANY_SOURCE`, one before and one after the `MPI_Bcast()` operation. The question is which message will be received from process 1 by which `MPI_Recv()`. Two execution orders are possible:

1. Process 1 first receives the message from process 2:

process 0		**process** 1		**process** 2
		`MPI_Recv()`	$\Longleftarrow$	`MPI_Send()`
`MPI_Bcast()`		`MPI_Bcast()`		`MPI_Bcast()`
`MPI_Send()`	$\Longrightarrow$	`MPI_Recv()`		

   This execution order may occur independent of whether system buffers are used or not. In particular, this execution order is possible also if the calls of `MPI_Bcast()` are synchronizing.

2. Process 1 first receives the message from process 0:

process 0		**process** 1		**process** 2
`MPI_Bcast()`				
`MPI_Send()`	$\Longrightarrow$	`MPI_Recv()`		
		`MPI_Bcast()`		
		`MPI_Recv()`	$\Longleftarrow$	`MPI_Send()`
				`MPI_Bcast()`

This execution order can only occur, if large enough system buffers are used, because otherwise process 0 cannot finish its `MPI_Bcast()` call before process 1 has started its corresponding `MPI_Bcast()`.

Thus, a non-deterministic program behavior results depending on the use of system buffers. Such a program is correct only if both execution orders lead to the intended result. The previous examples have shown that collective communication operations are synchronizing only if the MPI runtime system does not use system buffers to store messages locally before their actual transmission. Thus, when writing a parallel program, the programmer cannot rely on the expectation that collective communication operations lead to a synchronization of the participating processes.

To synchronize a group of processes, MPI provides the operation

```
MPI_Barrier (MPI_Comm comm).
```

The effect of this operation is that all processes belonging to the group of communicator `comm` are blocked until all other processes of this group also have called this operation.

5.3 Process Groups and Communicators

MPI allows the construction of subsets of processes by defining *groups* and *communicators*. A **process group** (or **group** for short) is an ordered set of processes of an application program. Each process of a group gets an uniquely defined process number which is also called **rank**. The ranks of a group always start with 0 and continue consecutively up to the number of processes minus one. A process may be a member of multiple groups and may have different ranks in each of these groups. The MPI system handles the representation and management of process groups. For the programmer, a group is an object of type `MPI_Group` which can only be accessed via a **handle** which may be internally implemented by the MPI system as an index or a reference. Process groups are useful for the implementation of **task-parallel programs** and are the basis for the communication mechanism of MPI.

In many situations, it is useful to partition the processes executing a parallel program into disjoint subsets (groups) which perform independent tasks of the program. This is called **task parallelism**, see also Sect. 3.3.4. The execution of task-parallel program parts can be obtained by letting the processes of a program call different functions or communication operations, depending on their process numbers. But task parallelism can be implemented much easier using the group concept.

5.3.1 Process Groups in MPI

MPI provides a lot of support for process groups. In particular, collective communication operations can be restricted to process groups by using the corresponding communicators. This is important for program libraries where the communication

operations of the calling application program and the communication operations of functions of the program library must be distinguished. If the same communicator is used, an error may occur, e.g., if the application program calls `MPI_Irecv()` with communicator `MPI_COMM_WORLD` using source `MPI_ANY_SOURCE` and tag `MPI_ANY_TAG` immediately before calling a library function. This is dangerous, if the library functions also use `MPI_COMM_WORLD` and if the library function called sends data to the process which executes `MPI_Irecv()` as mentioned above, since this process may then receive library-internal data. This can be avoided by using separate communicators.

In MPI, each point-to-point communication as well as each collective communication is executed in a **communication domain**. There is a separate communication domain for each process group using the ranks of the group. For each process of a group, the corresponding communication domain is *locally* represented by a **communicator**. In MPI, there is a communicator for each process group and each communicator defines a process group. A communicator knows all other communicators of the same communication domain. This may be required for the internal implementation of communication operations. Internally, a group may be implemented as an array of process numbers where each array entry specifies the global process number of one process of the group.

For the programmer, an MPI communicator is an opaque data object of type `MPI_Comm`. MPI distinguishes between **intra-communicators** and **inter-communicators**. Intra-communicators support the execution of arbitrary collective communication operations on a single group of processes. Inter-communicators support the execution of point-to-point communication operations between two process groups. In the following, we only consider intra-communicators which we call communicators for short.

In the preceding sections, we have always used the predefined communicator `MPI_COMM_WORLD` for communication. This communicator comprises all processes participating in the execution of a parallel program. MPI provides several operations to build additional process groups and communicators. These operations are all based on existing groups and communicators. The predefined communicator `MPI_COMM_WORLD` and the corresponding group are normally used as starting point. The process group to a given communicator can be obtained by calling

```
int MPI_Comm_group (MPI_Comm comm, MPI_Group *group),
```

where `comm` is the given communicator and `group` is a pointer to a previously declared object of type `MPI_Group` which will be filled by the MPI call. A predefined group is `MPI_GROUP_EMPTY` which denotes an empty process group.

5.3.1.1 Operations on Process Groups

MPI provides operations to construct new process groups based on existing groups. The predefined empty group `MPI_GROUP_EMPTY` can also be used. The **union** of two existing groups `group1` and `group2` can be obtained by calling

```
int MPI_Group_union (MPI_Group group1,
                     MPI_Group group2,
                     MPI_Group *new_group).
```

The ranks in the new group `new_group` are set such that the processes in `group1` keep their ranks. The processes from `group2` which are not in `group1` get subsequent ranks in consecutive order. The **intersection** of two groups is obtained by calling

```
int MPI_Group_intersection (MPI_Group group1,
                            MPI_Group group2,
                            MPI_Group *new_group),
```

where the process order from `group1` is kept for `new_group`. The processes in `new_group` get successive ranks starting from 0. The **set difference** of two groups is obtained by calling

```
int MPI_Group_difference (MPI_Group group1,
                          MPI_Group group2,
                          MPI_Group *new_group).
```

Again, the process order from `group1` is kept. A `sub_group` of an existing group can be obtained by calling

```
int MPI_Group_incl (MPI_Group group,
                    int p,
                    int *ranks,
                    MPI_Group *new_group),
```

where `ranks` is an integer array with `p` entries. The call of this function creates a new group `new_group` with `p` processes which have ranks from 0 to `p-1`. Process `i` is the process which has rank `ranks[i]` in the given group `group`. For a correct execution of this operation, `group` must contain at least `p` processes, and for $0 \leq$ `i` $<$ `p`, the values `ranks[i]` must be valid process numbers in `group` which are different from each other. Processes can be deleted from a given group by calling

```
int MPI_Group_excl (MPI_Group group,
                    int p,
                    int *ranks,
                    MPI_Group *new_group).
```

This function call generates a new group `new_group` which is obtained from `group` by deleting the processes with ranks `ranks[0]`, ..., `ranks[p-1]`. Again, the entries `ranks[i]` must be valid process ranks in `group` which are different from each other.

Data structures of type MPI_Group cannot be directly accessed by the programmer. But MPI provides operations to obtain information about process groups. The **size** of a process group can be obtained by calling

```
int MPI_Group_size (MPI_Group group, int *size),
```

where the size of the group is returned in parameter size. The **rank** of the calling process in a group can be obtained by calling

```
int MPI_Group_rank (MPI_Group group, int *rank),
```

where the rank is returned in parameter rank. The function

```
int MPI_Group_compare (MPI_Group group1, MPI_Group group2, int *res)
```

can be used to check whether two group representations group1 and group2 describe the same group. The parameter value res = MPI_IDENT is returned if both groups contain the same processes in the same order. The parameter value res = MPI_SIMILAR is returned if both groups contain the same processes, but group1 uses a different order than group2. The parameter value res = MPI_UNEQUAL means that the two groups contain different processes. The function

```
int MPI_Group_free (MPI_Group *group)
```

can be used to free a group representation if it is no longer needed. The group handle is set to MPI_GROUP_NULL.

5.3.1.2 Operations on Communicators

A new intra-communicator to a given group of processes can be generated by calling

```
int MPI_Comm_create (MPI_Comm comm,
                     MPI_Group group,
                     MPI_Comm *new_comm),
```

where comm specifies an existing communicator. The parameter group must specify a process group which is a subset of the process group associated with comm. For a correct execution, it is required that all processes of comm perform the call of MPI_Comm_create() and that each of these processes specifies the same group argument. As a result of this call, each calling process which is a member of group obtains a pointer to the new communicator in new_comm. Processes not belonging to group get MPI_COMM_NULL as return value in new_comm.

MPI also provides functions to get information about communicators. These functions are implemented as local operations which do not involve communication

to be executed. The size of the process group associated with a communicator comm can be requested by calling the function

```
int MPI_Comm_size (MPI_Comm comm, int *size).
```

The size of the group is returned in parameter size. For comm = MPI_COMM_WORLD the total number of processes executing the program is returned. The rank of a process in a particular group associated with a communicator comm can be obtained by calling

```
int MPI_Comm_rank (MPI_Comm comm, int *rank).
```

The group rank of the calling process is returned in rank. In previous examples, we have used this function to obtain the global rank of processes of MPI_COMM_WORLD. Two communicators comm1 and comm2 can be compared by calling

```
int MPI_Comm_compare (MPI_Comm comm1, MPI_Comm comm2, int *res).
```

The result of the comparison is returned in parameter res; res = MPI_IDENT is returned, if comm1 and comm2 denote the same communicator data structure. The value res = MPI_CONGRUENT is returned, if the associated groups of comm1 and comm2 contain the same processes with the same rank order. If the two associated groups contain the same processes in different rank order, res = MPI_SIMILAR is returned. If the two groups contain different processes, res = MPI_UNEQUAL is returned.

For the direct construction of communicators, MPI provides operations for the duplication, deletion, and splitting of communicators. A communicator can be **duplicated** by calling the function

```
int MPI_Comm_dup (MPI_Comm comm, MPI_Comm *new_comm),
```

which creates a new intra-communicator new_comm with the same characteristics (assigned group and topology) as comm. The new communicator new_comm represents a new distinct communication domain. Duplicating a communicator allows the programmer to separate communication operations executed by a library from communication operations executed by the application program itself, thus avoiding any conflict. A communicator can be **deallocated** by calling the MPI operation

```
int MPI_Comm_free (MPI_Comm *comm).
```

This operation has the effect that the communicator data structure comm is freed as soon as all pending communication operations performed with this communicator are completed. This operation could, e.g., be used to free a communicator which has previously been generated by duplication to separate library communication from

communication of the application program. Communicators should not be assigned by simple assignments of the form `comm1 = comm2`, since a deallocation of one of the two communicators involved with `MPI_Comm_free()` would have a side effect on the other communicator, even if this is not intended. A **splitting** of a communicator can be obtained by calling the function

```
int MPI_Comm_split (MPI_Comm comm,
                    int color,
                    int key,
                    MPI_Comm *new_comm).
```

The effect is that the process group associated with `comm` is partitioned into disjoint subgroups. The number of subgroups is determined by the number of different values of `color`. Each subgroup contains all processes which specify the same value for `color`. Within each subgroup, the processes are ranked in the order defined by argument value `key`. If two processes in a subgroup specify the same value for `key`, the order in the original group is used. If a process of `comm` specifies `color = MPI_UNDEFINED`, it is not a member of any of the subgroups generated. The subgroups are not directly provided in the form of an `MPI_GROUP` representation. Instead, each process of `comm` gets a pointer `new_comm` to the communicator of that subgroup which the process belongs to. For `color = MPI_UNDEFINED`, `MPI_COMM_NULL` is returned as `new_comm`.

Example We consider a group of 10 processes each of which calls the operation `MPI_Comm_split()` with the following argument values [163]:

process	a	b	c	d	e	f	g	h	i	j
rank	0	1	2	3	4	5	6	7	8	9
color	0	$\perp$	3	0	3	0	0	5	3	$\perp$
key	3	1	2	5	1	1	1	2	1	0

This call generates three subgroups {f, g, a, d}, {e, i, c}, and {h} which contain the processes in this order. In the table, the entry $\perp$ represents `color = MPI_UNDEFINED`. □

The operation `MPI_Comm_split()` can be used to prepare a task-parallel execution. The different communicators generated can be used to perform communication within the task-parallel parts, thus separating the communication domains.

5.3.2 Process Topologies

Each process of a process group has a unique rank within this group which can be used for communication with this process. Although a process is uniquely defined by its group rank, it is often useful to have an alternative representation and access. This is the case if an algorithm performs computations and communication on a two-dimensional or a three-dimensional grid where grid points are assigned to different

processes and the processes exchange data with their neighboring processes in each dimension by communication. In such situations, it is useful if the processes can be arranged according to the communication pattern in a grid structure such that they can be addressed via two-dimensional or three-dimensional coordinates. Then each process can easily address its neighboring processes in each dimension. MPI supports such a logical arrangement of processes by defining **virtual topologies** for intra-communicators, which can be used for communication within the associated process group.

A virtual Cartesian grid structure of arbitrary dimension can be generated by calling

```
int MPI_Cart_create (MPI_Comm comm,
                     int ndims,
                     int *dims,
                     int *periods,
                     int reorder,
                     MPI_Comm *new_comm)
```

where `comm` is the original communicator without topology, `ndims` specifies the number of dimensions of the grid to be generated, `dims` is an integer array of size `ndims` such that `dims[i]` is the number of processes in dimension `i`. The entries of `dims` must be set such that the product of all entries is the number of processes contained in the new communicator `new_comm`. In particular, this product must not exceed the number of processes of the original communicator `comm`. The boolean array `periods` of size `ndims` specifies for each dimension whether the grid is periodic (entry 1 or `true`) or not (entry 0 or `false`) in this dimension. For `reorder = false`, the processes in `new_comm` have the same rank as in `comm`. For `reorder = true`, the runtime system is allowed to reorder processes, e.g., to obtain a better mapping of the process topology to the physical network of the parallel machine.

Example We consider a communicator with 12 processes [163]. For `ndims=2`, using the initializations `dims[0]=3`, `dims[1]=4`, `periods[0]=periods[1]=0`, `reorder=0`, the call

```
MPI_Cart_create (comm, ndims, dims, periods, reorder, &new_comm)
```

generates a virtual 3×4 grid with the following group ranks and coordinates:

0 (0,0)	1 (0,1)	2 (0,2)	3 (0,3)
4 (1,0)	5 (1,1)	6 (1,2)	7 (1,3)
8 (2,0)	9 (2,1)	10 (2,2)	11 (2,3)

The Cartesian coordinates are represented in the form (row, column). In the communicator, the processes are ordered according to their rank rowwise in increasing order. □

To help the programmer to select a balanced distribution of the processes for the different dimensions, MPI provides the function

```
int MPI_Dims_create (int nnodes, int ndims, int *dims)
```

where `ndims` is the number of dimensions in the grid and `nnodes` is the total number of processes available. The parameter `dims` is an integer array of size `ndims`. After the call, the entries of `dims` are set such that the `nnodes` processes are balanced as much as possible among the different dimensions, i.e., each dimension has about equal size. But the size of a dimension `i` is set only if `dims[i] = 0` when calling `MPI_Dims_create()`. The number of processes in a dimension `j` can be fixed by setting `dims[j]` to a positive value before the call. This entry is then not modified by this call and the other entries of `dims` are set by the call accordingly.

When defining a virtual topology, each process has a group rank, and also a position in the virtual grid topology which can be expressed by its Cartesian coordinates. For the translation between group ranks and Cartesian coordinates, MPI provides two operations. The operation

```
int MPI_Cart_rank (MPI_Comm comm, int *coords, int *rank)
```

translates the Cartesian coordinates provided in the integer array `coords` into a group rank and returns it in parameter `rank`. The parameter `comm` specifies the communicator with Cartesian topology. For the opposite direction, the operation

```
int MPI_Cart_coords (MPI_Comm comm,
                     int rank,
                     int ndims,
                     int *coords)
```

translates the group rank provided in `rank` into Cartesian coordinates, returned in integer array `coords`, for a virtual grid; `ndims` is the number of dimensions of the virtual grid defined for communicator `comm`.

Virtual topologies are typically defined to facilitate the determination of communication partners of processes. A typical communication pattern in many grid-based algorithms is that processes communicate with their neighboring processes in a specific dimension. To determine these neighboring processes, MPI provides the operation

```
int MPI_Cart_shift (MPI_Comm comm,
                    int dir,
                    int displ,
                    int *rank_source,
                    int *rank_dest)
```

where dir specifies the dimension for which the neighboring process should be determined. The parameter displ specifies the displacement, i.e., the distance to the neighbor. Positive values of displ request the neighbor in upward direction, negative values request for downward direction. Thus, displ = -1 requests the neighbor immediately preceding, displ = 1 requests the neighboring process which follows directly. The result of the call is that rank_dest contains the group rank of the neighboring process in the specified dimension and distance. The rank of the process for which the calling process is the neighboring process in the specified dimension and distance is returned in rank_source. Thus, the group ranks returned in rank_dest and rank_source can be used as parameters for MPI_Sendrecv(), as well as for separate MPI_Send() and MPI_Recv(), respectively.

Example As example, we consider 12 processes that are arranged in a 3×4 grid structure with periodic connections [163]. Each process stores a floating-point value which is exchanged with the neighboring process in dimension 0, i.e., within the columns of the grid:

```
int coords[2], dims[2], periods[2], source, dest, my_rank,
    reorder;
MPI_Comm comm_2d;
MPI_status status;
float a, b;
MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);
dims[0] = 3; dims[1] = 4;
periods[0] = periods[1] = 1;
reorder = 0;
MPI_Cart_create (MPI_COMM_WORLD, 2, dims, periods, reorder,
   &comm_2d);
MPI_Cart_coords (comm_2d, my_rank, 2, coords);
MPI_Cart_shift (comm_2d, 0, coords[1], &source, &dest);
a = my_rank;
MPI_Sendrecv (&a, 1, MPI_FLOAT, dest, 0, &b, 1, MPI_FLOAT,
   source, 0, comm_2d, &status);
```

In this example, the specification displs = coord[1] is used as displacement for MPI_Cart_shift(), i.e., the position in dimension 1 is used as displacement. Thus, the displacement increases with column position, and in each column of the grid, a different exchange is executed. MPI_Cart_shift() is used to determine the communication partners dest and source for each process. These are then used as parameters for MPI_Sendrecv(). The following diagram illustrates the exchange. For each process, its rank, its Cartesian coordinates, and its communication partners in the form source/dest are given in this order. For example, for the process with rank=5, it is coords[1]=1, and therefore source=9 (lower neighbor in dimension 0) and dest=1 (upper neighbor in dimension 0).

□

0 (0,0) 0|0	1 (0,1) 9|5	2 (0,2) 6|10	3 (0,3) 3|3
4 (1,0) 4|4	5 (1,1) 1|9	6 (1,2) 10|2	7 (1,3) 7|7
8 (2,0) 8|8	9 (2,1) 5|1	10 (2,2) 2|6	11 (2,3) 11|11

If a virtual topology has been defined for a communicator, the corresponding grid can be partitioned into subgrids by using the MPI function

```
int MPI_Cart_sub (MPI_Comm comm,
                  int *remain_dims,
                  MPI_Comm *new_comm).
```

The parameter `comm` denotes the communicator for which the virtual topology has been defined. The subgrid selection is controlled by the integer array `remain_dims` which contains an entry for each dimension of the original grid.

Setting `remain_dims[i] = 1` means that the `i`th dimension is kept in the subgrid; `remain_dims[i] = 0` means that the `i`th dimension is dropped in the subgrid. In this case, the size of this dimension determines the number of subgrids generated in this dimension. A call of `MPI_Cart_sub()` generates a new communicator `new_comm` for each calling process, representing the corresponding subgroup of the subgrid to which the calling process belongs. The dimensions of the different subgrids result from the dimensions for which `remain_dims[i]` has been set to `1`. The total number of subgrids generated is defined by the product of the number of processes in all dimensions `i` for which `remain_dims[i]` has been set to `0`.

Example We consider a communicator `comm` for which a $2 \times 3 \times 4$ virtual grid topology has been defined. Calling

```
int MPI_Cart_sub (comm_3d, remain_dims, &new_comm)
```

with `remain_dims=(1,0,1)` generates three 2×4 grids and each process gets a communicator for its corresponding subgrid, see Fig. 5.12 for an illustration. □

MPI also provides functions to inquire information about a virtual topology that has been defined for a communicator. The MPI function

```
int MPI_Cartdim_get (MPI_Comm comm,int *ndims)
```

returns in parameter `ndims` the number of dimensions of the virtual grid associated with communicator `comm`. The MPI function

```
int MPI_Cart_get (MPI_Comm comm,
                  int maxdims,
                  int *dims,
                  int *periods,
                  int *coords)
```

returns information about the virtual topology defined for communicator `comm`. This virtual topology should have `maxdims` dimensions, and the arrays `dims`, `periods`, and `coords` should have this size. The following information is returned by this call: Integer array `dims` contains the number of processes in each dimension of the virtual grid, the boolean array `periods` contains the corresponding periodicity information. The integer array `coords` contains the Cartesian coordinates of the calling process.

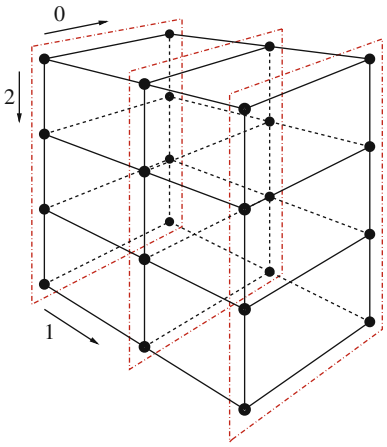


Fig. 5.12 Partitioning of a three-dimensional grid of size $2 \times 3 \times 4$ into three two-dimensional grids of size 2×4 each

This figure will be printed in b/w

5.3.3 Timings and Aborting Processes

To measure the parallel execution times of program parts, MPI provides the function

```
double MPI_Wtime (void)
```

which returns as a floating-point value the number of seconds elapsed since a fixed point in time in the past. A typical usage for timing would be:

```
start = MPI_Wtime();
part_to_measure();
end = MPI_Wtime();
```

`MPI_Wtime()` does not return a system time, but the absolute time elapsed between the start and the end of a program part, including times at which the

process executing `part_to_measure()` has been interrupted. The resolution of `MPI_Wtime()` can be requested by calling

```
double MPI_Wtick (void)
```

which returns the time between successive clock ticks in seconds as floating-point value. If the resolution is a microsecond, `MPI_Wtick()` will return 10^{-6}. The execution of all processes of a communicator can be aborted by calling the MPI function

```
int MPI_Abort (MPI_Comm comm, int error_code)
```

where `error_code` specifies the error code to be used, i.e., the behavior is as if the main program has been terminated with `return error_code`.

5.4 Introduction to MPI-2

For a continuous development of MPI, the MPI Forum has defined extensions to MPI as described in the previous sections. These extensions are often referred to as MPI-2. The original MPI standard is referred to as MPI-1. The current version of MPI-1 is described in the MPI document, version 1.3 [55]. Since MPI-2 comprises all MPI-1 operations, each correct MPI-1 program is also a correct MPI-2 program. The most important extensions contained in MPI-2 are dynamic process management, one-sided communications, parallel I/O, and extended collective communications. In the following, we give a short overview of the most important extensions. For a more detailed description, we refer to the current version of the MPI-2 document, version 2.1, see [56].

5.4.1 Dynamic Process Generation and Management

MPI-1 is based on a **static process model**: The processes used for the execution of a parallel program are implicitly created before starting the program. No processes can be added during program execution. Inspired by PVM [63], MPI-2 extends this process model to a **dynamic process model** which allows the creation and deletion of processes at any time during program execution. MPI-2 defines the interface for dynamic process management as a collection of suitable functions and gives some advice for an implementation. But not all implementation details are fixed to support an implementation for different operating systems.

5.4.1.1 `MPI_Info` Objects

Many MPI-2 functions use an additional argument of type `MPI_Info` which allows the provision of additional information for the function, depending on the spe-

cific operating system used. But using this feature may lead to non-portable MPI programs. MPI_Info provides opaque objects where each object can store arbitrary (key, value) pairs. In C, both entries are strings of type char, terminated with \0. Since MPI_Info objects are opaque, their implementation is hidden from the user. Instead, some functions are provided for access and manipulation. The most important ones are described in the following. The function

```
int MPI_Info_create (MPI_Info *info)
```

can be used to generate a new object of type MPI_Info. Calling the function

```
int MPI_Info_set (MPI_Info info, char *key, char *value)
```

adds a new (key, value) pair to the MPI_Info structure info. If a value for the same key was previously stored, the old value is overwritten. The function

```
int MPI_Info_get (MPI_Info info,
                  char *key,
                  int valuelen,
                  char *value,
                  int *flag)
```

can be used to retrieve a stored pair (key, value) from info. The programmer specifies the value of key and the maximum length valuelen of the value entry. If the specified key exists in info, the associated value is returned in parameter value. If the associated value string is longer than valuelen, the returned string is truncated after valuelen characters. If the specified key exists in info, true is returned in parameter flag; otherwise, false is returned. The function

```
int MPI_Info_delete(MPI_Info info, char *key)
```

can be used to delete an entry (key, value) from info. Only the key has to be specified.

5.4.1.2 Process Creation and Management

A number of MPI processes can be started by calling the function

```
int MPI_Comm_spawn (char *command,
                    char *argv[],
                    int maxprocs,
                    MPI_Info info,
                    int root,
                    MPI_Comm comm,
                    MPI_Comm *intercomm,
                    int errcodes[]).
```

The parameter command specifies the name of the program to be executed by each of the processes, argv[] contains the arguments for this program. In contrast to the standard C convention, argv[0] is not the program name but the first argument for the program. An empty argument list is specified by MPI_ARGV_NULL. The parameter maxprocs specifies the number of processes to be started. If the MPI runtime system is not able to start maxprocs processes, an error message is generated. The parameter info specifies an MPI_Info data structure with (key, value) pairs providing additional instructions for the MPI runtime system on how to start the processes. This parameter could be used to specify the path of the program file as well as its arguments, but this may lead to non-portable programs. Portable programs should use MPI_INFO_NULL.

The parameter root specifies the number of the root process from which the new processes are spawned. Only this root process provides values for the preceding parameters. But the function MPI_Comm_spawn() is a collective operation, i.e., all processes belonging to the group of the communicator comm must call the function. The parameter intercomm contains an intercommunicator after the successful termination of the function call. This intercommunicator can be used for communication between the original group of comm and the group of processes just spawned.

The parameter errcodes is an array with maxprocs entries in which the status of each process to be spawned is reported. When a process could be spawned successfully, its corresponding entry in errcodes will be set to MPI_SUCCESS. Otherwise, an implementation-specific error code will be reported.

A successful call of MPI_Comm_spawn() starts maxprocs identical copies of the specified program and creates an intercommunicator, which is provided to all calling processes. The new processes belong to a separate group and have a separate MPI_COMM_WORLD communicator comprising all processes spawned. The spawned processes can access the intercommunicator created by MPI_Comm_spawn() by calling the function

```
int MPI_Comm_get_parent(MPI_Comm *parent).
```

The requested intercommunicator is returned in parameter parent. Multiple MPI programs or MPI programs with different argument values can be spawned by calling the function

```
int MPI_Comm_spawn_multiple (int count,
                             char *commands[],
                             char **argv[],
                             int maxprocs[],
                             MPI_Info infos[],
                             int root,
                             MPI_Comm comm,
                             MPI_Comm *intercomm,
                             int errcodes[])
```

where count specifies the number of different programs to be started. Each of the following four arguments specifies an array with count entries where each entry has the same type and meaning as the corresponding parameters for MPI_Comm_spawn(): The argument commands[] specifies the names of the programs to be started, argv[] contains the corresponding arguments, maxprocs[] defines the number of copies to be started for each program, and infos[] provides additional instructions for each program. The other arguments have the same meaning as for MPI_comm_spawn().

After the call of MPI_Comm_spawn_multiple() has been terminated, the array errcodes[] contains an error status entry for each process created. The entries are arranged in the order given by the commands[] array. In total, errcodes[] contains

$$\sum_{i=0}^{\texttt{count}-1} \texttt{maxprocs[i]}$$

entries. There is a difference between calling MPI_Comm_spawn() multiple times and calling MPI_Comm_spawn_multiple() with the same arguments. Calling the function MPI_Comm_spawn_multiple() creates one communicator MPI_COMM_WORLD for all newly created processes. Multiple calls of MPI_Comm_spawn() generate separate communicators MPI_COMM_WORLD, one for each process group created.

The attribute MPI_UNIVERSE_SIZE specifies the maximum number of processes that can be started in total for a given application program. The attribute is initialized by MPI_Init().

5.4.2 One-Sided Communication

MPI provides single transfer and collective communication operations as described in the previous sections. For collective communication operations, each process of a communicator calls the communication operation to be performed. For single-transfer operations, a sender and a receiver process must cooperate and actively execute communication operations: In the simplest case, the sender executes an MPI_Send() operation, and the receiver executes an MPI_Recv() operation. Therefore, this form of communication is also called *two-sided communication*. The position of the MPI_Send() operation in the sender process determines at which time the data is sent. Similarly, the position of the MPI_Recv() operation in the receiver process determines at which time the receiver stores the received data in its local address space.

In addition to two-sided communication, MPI-2 supports *one-sided communication*. Using this form of communication, a source process can access the address space at a target process without an active participation of the target process. This form of communication is also called Remote Memory Access (RMA). RMA facilitates communication for applications with dynamically changing data access

patterns by supporting a flexible dynamic distribution of program data among the address spaces of the participating processes. But the programmer is responsible for the coordinated memory access. In particular, a concurrent manipulation of the same address area by different processes at the same time must be avoided to inhibit race conditions. Such race conditions cannot occur for two-sided communications.

5.4.2.1 Window Objects

If a process A should be allowed to access a specific memory region of a process B using one-sided communication, process B must expose this memory region for external access. Such a memory region is called *window*. A window can be exposed by calling the function

```
int MPI_Win_create (void *base,
                    MPI_Aint size,
                    int displ_unit,
                    MPI_Info info,
                    MPI_Comm comm,
                    MPI_Win *win).
```

This is a collective call which must be executed by each process of the communicator `comm`. Each process specifies a window in its local address space that it exposes for RMA by other processes of the same communicator.

The starting address of the window is specified in parameter `base`. The size of the window is given in parameter `size` as number of bytes. For the size specification, the predefined MPI type `MPI_Aint` is used instead of `int` to allow window sizes of more than 2^{32} bytes. The parameter `displ_unit` specifies the displacement (in bytes) between neighboring window entries used for one-sided memory accesses. Typically, `displ_unit` is set to `1` if bytes are used as unit or to `sizeof(type)` if the window consists of entries of type `type`. The parameter `info` can be used to provide additional information for the runtime system. Usually, `info=MPI_INFO_NULL` is used. The parameter `comm` specifies the communicator of the processes which participate in the `MPI_Win_create()` operation. The call of `MPI_Win_create()` returns a window object of type `MPI_Win` in parameter `win` to the calling process. This window object can then be used for RMA to memory regions of other processes of `comm`.

A window exposed for external accesses can be closed by letting all processes of the corresponding communicator call the function

```
int MPI_Win_free (MPI_Win *win)
```

thus freeing the corresponding window object `win`. Before calling `MPI_Win_free()`, the calling process must have finished all operations on the specified window.

5.4.2.2 RMA Operations

For the actual one-sided data transfer, MPI provides three *non-blocking* RMA operations: `MPI_Put()` transfers data from the memory of the calling process into the window of another process; `MPI_Get()` transfers data from the window of a target process into the memory of the calling process; `MPI_Accumulate()` supports the accumulation of data in the window of the target process. These operations are *non-blocking*: When control is returned to the calling process, this does not necessarily mean that the operation is completed. To test for the completion of the operation, additional synchronization operations like `MPI_Win_fence()` are provided as described below. Thus, a similar usage model as for non-blocking two-sided communication can be used. The local buffer of an RMA communication operation should not be updated or accessed until the subsequent synchronization call returns.

The transfer of a data block into the window of another process can be performed by calling the function

```
int MPI_Put (void *origin_addr,
             int origin_count,
             MPI_Datatype origin_type,
             int target_rank,
             MPI_Aint target_displ,
             int target_count,
             MPI_Datatype target_type,
             MPI_Win win)
```

where `origin_addr` specifies the start address of the data buffer provided by the calling process and `origin_count` is the number of buffer entries to be transferred. The parameter `origin_type` defines the type of the entries. The parameter `target_rank` specifies the rank of the target process which should receive the data block. This process must have created the window object `win` by a preceding `MPI_Win_create()` operation, together with all processes of the communicator group to which the process calling `MPI_Put()` also belongs to. The remaining parameters define the position and size of the target buffer provided by the target process in its window: `target_displ` defines the displacement from the start of the window to the start of the target buffer, `target_count` specifies the number of entries in the target buffer, `target_type` defines the type of each entry in the target buffer. The data block transferred is stored in the memory of the target process at position `target_addr` := `window_base + target_displ * displ_unit` where `window_base` is the start address of the window in the memory of the target process and `displ_unit` is the distance between neighboring window entries as defined by the target process when creating the window with `MPI_Win_create()`. The execution of an `MPI_Put()` operation by a process `source` has the same effect as a two-sided communication for which process `source` executes the send operation

```
int MPI_Isend (origin_addr, origin_count, origin_type,
               target_rank, tag, comm)
```

and the target process executes the receive operation

```
int MPI_Recv (target_addr, target_count, target_type,
              source, tag, comm, &status)
```

where comm is the communicator for which the window object has been defined. For a correct execution of the operation, some constraints must be satisfied: The target buffer defined must fit in the window of the target process and the data block provided by the calling process must fit into the target buffer. In contrast to MPI_Isend() operations, the send buffers of multiple successive MPI_Put() operations may overlap, even if there is no synchronization in between. Source and target processes of an MPI_Put() operation may be identical.

To transfer a data block from the window of another process into a local data buffer, the MPI function

```
int MPI_Get (void *origin_addr,
             int origin_count,
             MPI_Datatype origin_type,
             int target_rank,
             MPI_Aint target_displ,
             int target_count,
             MPI_Datatype target_type,
             MPI_Win win)
```

is provided. The parameter origin_addr specifies the start address of the receive buffer in the local memory of the calling process; origin_count defines the number of elements to be received; origin_type is the type of each of the elements. Similar to MPI_Put(), target_rank specifies the rank of the target process which provides the data and win is the window object previously created. The remaining parameters define the position and size of the data block to be transferred out of the window of the target process. The start address of the data block in the memory of the target process is given by target_addr := window_base + target_displ * displ_unit.

For the accumulation of data values in the memory of another process, MPI provides the operation

```
int MPI_Accumulate (void *origin_addr,
                    int origin_count,
                    MPI_Datatype origin_type,
                    int target_rank,
                    MPI_Aint target_displ,
                    int target_count,
                    MPI_Datatype target_type,
                    MPI_Op op,
                    MPI_Win win)
```

The parameters have the same meaning as for MPI_Put(). The additional parameter op specifies the reduction operation to be applied for the accumulation. The same predefined reduction operations as for MPI_Reduce() can be used, see Sect. 5.2, p. 215. Examples are MPI_MAX and MPI_SUM. User-defined reduction operations cannot be used. The execution of an MPI_Accumulate() has the effect that the specified reduction operation is applied to corresponding entries of the source buffer and the target buffer and that the result is written back into the target buffer. Thus, data values can be accumulated in the target buffer provided by another process. There is an additional reduction operation MPI_REPLACE which allows the replacement of buffer entries in the target buffer, without taking the previous values of the entries into account. Thus, MPI_Put() can be considered as a special case of MPI_Accumulate() with reduction operation MPI_REPLACE.

There are some constraints for the execution of one-sided communication operations by different processes to avoid race conditions and to support an efficient implementation of the operations. Concurrent conflicting accesses to the same memory location in a window are not allowed. At each point in time during program execution, each memory location of a window can be used as target of at most one one-sided communication operation. Exceptions are accumulation operations: Multiple concurrent MPI_Accumulate() operations can be executed at the same time for the same memory location. The result is obtained by using an arbitrary order of the executed accumulation operations. The final accumulated value is the same for all orders, since the predefined reduction operations are commutative. A window of a process P cannot be used concurrently by an MPI_Put() or MPI_Accumulate() operation of another process and by a local store operation of P, even if different locations in the window are addressed.

MPI provides three synchronization mechanisms for the coordination of one-sided communication operations executed in the windows of a group of processes. These three mechanisms are described in the following.

5.4.2.3 Global Synchronization

A global synchronization of all processes of the group of a window object can be obtained by calling the MPI function

```
int MPI_Win_fence (int assert, MPI_Win win)
```

where win specifies the window object. MPI_Win_fence() is a collective operation to be performed by all processes of the group of win. The effect of the call is that all RMA operations originating from the calling process and started before the MPI_Win_fence() call are locally completed at the calling process before control is returned to the calling process. RMA operations started after the MPI_Win_fence() call accesses the specified target window only after the corresponding target process has called its corresponding MPI_Win_fence() operation. The intended use of MPI_Win_fence() is the definition of program areas in which one-sided communication operations are executed. Such program areas

are surrounded by calls of MPI_Win_fence(), thus establishing communication phases that can be mixed with computation phases during which no communication is required. Such communication phases are also referred to as **access epochs** in MPI. The parameter assert can be used to specify assertions on the context of the call of MPI_Win_fence() which can be used for optimizations by the MPI runtime system. Usually, assert=0 is used, not providing additional assertions.

Global synchronization with MPI_Win_fence() is useful in particular for applications with regular communication pattern in which computation phases alternate with communication phases.

Example As example, we consider an iterative computation of a distributed data structure A. In each iteration step, each participating process updates its local part of the data structure using the function update(). Then, parts of the local data structure are transferred into the windows of neighboring processes using MPI_Put(). Before the transfer, the elements to be transferred are copied into a contiguous buffer. This copy operation is performed by update_buffer(). The communication operations are surrounded by MPI_Win_fence() operations to separate the communication phases of successive iterations from each other. This results in the following program structure:

```
while (!converged(A)) {
   update(A);
   update_buffer(A, from_buf);
   MPI_Win_fence(0, win);
   for (i=0; i<num_neighbors; i++)
      MPI_Put(&from_buf[i], size[i], MPI_INT, neighbor[i],
      to_disp[i],
              size[i], MPI_INT, win);
   MPI_Win_fence(0, win);
}
```

The iteration is controlled by the function converged().

5.4.2.4 Loose Synchronization

MPI also supports a loose synchronization which is restricted to pairs of communicating processes. To perform this form of synchronization, an accessing process defines the start and the end of an **access epoch** by a call to MPI_Win_start() and MPI_Win_complete(), respectively. The target process of the communication defines a corresponding **exposure epoch** by calling MPI_Win_post() to start the exposure epoch and MPI_Win_wait() to end the exposure epoch. A synchronization is established between MPI_Win_start() and MPI_Win_post() in the sense that all RMAs which the accessing process issues after its MPI_Win_start() call are executed not before the target process has completed its MPI_Win_post() call. Similarly, a synchronization between MPI_Win_complete() and MPI_Win_wait() is established in the sense that the MPI_Win_wait() call is

completed at the target process not before all RMAs of the accessing process in the corresponding access epoch are terminated.

To use this form of synchronization, before performing an RMA, a process defines the start of an access epoch by calling the function

```
int MPI_Win_start (MPI_Group group,
                   int assert,
                   MPI_Win win)
```

where `group` is a group of target processes. Each of the processes in `group` must issue a matching call of `MPI_Win_post()`. The parameter `win` specifies the window object to which the RMA is made. MPI supports a blocking and a non-blocking behavior of `MPI_Win_start()`:

- Blocking behavior: The call of `MPI_Win_start()` is blocked until all processes of `group` have completed their corresponding calls of `MPI_Win_post()`.
- Non-blocking behavior: The call of `MPI_Win_start()` is completed at the accessing process without blocking, even if there are processes in `group` which have not yet issued or finished their corresponding call of `MPI_Win_post()`. Control is returned to the accessing process and this process can issue RMA operations like `MPI_Put()` or `MPI_Get()`. These calls are then delayed until the target process has finished its `MPI_Win_post()` call.

The exact behavior depends on the MPI implementation. The end of an access epoch is indicated by the accessing process by calling

```
int MPI_Win_complete (MPI_Win win)
```

where `win` is the window object which has been accessed during this access epoch. Between the call of `MPI_Win_start()` and `MPI_Win_complete()`, only RMA operations to the window `win` of processes belonging to `group` are allowed. When calling `MPI_Win_complete()`, the calling process is blocked until all RMA operations to `win` issued in the corresponding access epoch have been completed at the accessing process. An `MPI_Put()` call issued in the access epoch can be completed at the calling process as soon as the local data buffer provided can be reused. But this does not necessarily mean that the data buffer has already been stored in the window of the target process. It might as well have been stored in a local system buffer of the MPI runtime system. Thus, the termination of `MPI_Win_complete()` does not imply that all RMA operations have taken effect at the target processes.

A process indicates the start of an RMA exposure epoch for a local window `win` by calling the function

```
int MPI_Win_post (MPI_Group group,
                  int assert,
                  MPI_Win win).
```

Only processes in group are allowed to access the window during this exposure epoch. Each of the processes in group must issue a matching call of the function MPI_Win_start(). The call of MPI_Win_post() is non-blocking. A process indicates the end of an RMA exposure epoch for a local window win by calling the function

```
int MPI_Win_wait (MPI_Win win).
```

This call blocks until all processes of the group defined in the corresponding MPI_Win_post() call have issued their corresponding MPI_Win_complete() calls. This ensures that all these processes have terminated the RMA operations of their corresponding access epoch to the specified window. Thus, after the termination of MPI_Win_wait(), the calling process can reuse the entries of its local window, e.g., by performing local accesses. During an exposure epoch, indicated by surrounding MPI_Win_post() and MPI_Win_wait() calls, a process should not perform local operations on the specified window to avoid access conflicts with other processes.

By calling the function

```
int MPI_Win_test (MPI_Win win, int *flag)
```

a process can test whether the RMA operation of other processes to a local window has been completed or not. This call can be considered as the non-blocking version of MPI_Win_wait(). The parameter flag=1 is returned by the call if all RMA operations to win have been terminated. In this case, MPI_Win_test() has the same effect as MPI_Win_wait() and should not be called again for the same exposure epoch. The parameter flag=0 is returned if not all RMA operations to win have been finished yet. In this case, the call has no further effect and can be repeated later.

The synchronization mechanism described can be used for arbitrary communication patterns on a group of processes. A communication pattern can be described by a directed graph $G = (V, E)$ where V is the set of participating processes. There exists an edge $(i, j) \in E$ from process i to process j, if i accesses the window of j by an RMA operation. Assuming that the RMA operations are performed on window win, the required synchronization can be reached by letting each participating process execute MPI_Win_start(target_group,0,win) followed by MPI_Win_post(source_group,0,win) where source_group= $\{i;(i, j) \in E\}$ denotes the set of accessing processes and target_group= $\{j;(i, j) \in E\}$ denotes the set of target processes.

Example This form of synchronization is illustrated by the following example, which is a variation of the previous example describing the iterative computation of a distributed data structure:

```
while (!converged (A)) {
   update(A);
   update_buffer(A, from_buf);
   MPI_Win_start(target_group, 0, win);
   MPI_Win_post(source_group, 0, win);
   for (i=0; i<num_neighbors; i++)
      MPI_Put(&from_buf[i], size[i], MPI_INT, neighbor[i], to
      _disp[i],
              size[i], MPI_INT, win);
   MPI_Win_complete(win);
   MPI_Win_wait(win);
}
```

In the example, it is assumed that `source_group` and `target_group` have been defined according to the communication pattern used by all processes as described above. An alternative would be that each process defines a set `source_group` of processes which are allowed to access its local window and a set `target_group` of processes whose window the process is going to access. Thus, each process potentially defines different source and target groups, leading to a weaker form of synchronization as for the case that all processes define the same source and target groups. □

5.4.2.5 Lock Synchronization

To support the model of a shared address space, MPI provides a synchronization mechanism for which only the accessing process actively executes communication operations. Using this form of synchronization, it is possible that two processes exchange data via RMA operations executed on the window of a third process without an active participation of the third process. To avoid access conflicts, a lock mechanism is provided as typically used in programming environments for shared address spaces, see Chap. 6. This means that the accessing process locks the accessed window before the actual access and releases the lock again afterwards. To lock a window before an RMA operation, MPI provides the operation

```
int MPI_Win_lock (int lock_type,
                  int rank,
                  int assert,
                  MPI_Win win).
```

A call of this function starts an RMA access epoch for the window `win` at the process with rank `rank`. Two lock types are supported, which can be specified by parameter `lock_type`. An *exclusive lock* is indicated by `lock_type = MPI_LOCK_EXCLUSIVE`. This lock type guarantees that the following RMA operations executed by the calling process are protected from RMA operations of other processes, i.e., exclusive access to the window is ensured. Exclusive locks should

be used if the executing process will change the value of window entries using MPI_Put() and if these entries could also be accessed by other processes.

A shared lock is indicated by lock_type = MPI_LOCK_SHARED. This lock type guarantees that the following RMA operations of the calling process are protected from *exclusive* RMA operations of other processes, i.e., other processes are not allowed to change entries of the window via RMA operations that are protected by an exclusive lock. But other processes are allowed to perform RMA operations on the same window that are also protected by a shared lock.

Shared locks should be used if the executing process accesses window entries only by MPI_Get() or MPI_Accumulate(). When a process wants to read or manipulate entries of its local window using local operations, it must protect these local operations with a lock mechanism, if these entries can also be accessed by other processes.

An access epoch started by MPI_Win_lock() for a window win can be terminated by calling the MPI function

```
int MPI_Win_unlock (int rank,
                    MPI_Win win)
```

where rank is the rank of the target process. The call of this function blocks until all RMA operations issued by the calling process on the specified window have been completed both at the calling process and at the target process. This guarantees that all manipulations of window entries issued by the calling process have taken effect at the target process.

Example The use of lock synchronization for the iterative computation of a distributed data structure is illustrated in the following example which is a variation of the previous examples. Here, an exclusive lock is used to protect the RMA operations:

```
while (!converged (A)) {
   update(A);
   update_buffer(A, from_buf);
   MPI_Win_start(target_group, 0, win);
   for (i=0; i<num_neighbors; i++) {
      MPI_Win_lock(MPI_LOCK_EXCLUSIVE, neighbor[i], 0, win);
      MPI_Put(&from_buf[i], size[i], MPI_INT, neighbor[i], to
      _disp[i],
              size[i], MPI_INT, win);
      MPI_Win_unlock(neighbor[i], win);
   }
}
```

5.5 Exercises for Chap. 5

Exercise 5.1 Consider the following incomplete piece of an MPI program:

```
int rank, p, size=8;
int left, right;
char send_buffer1[8], recv_buffer1[8];
char send_buffer2[8], recv_buffer2[8];
...
MPI_Comm_rank(MPI_COMM_WORLD, 8 rank);
MPI_Comm_size(MPI_COMM_WORLD, & p);
left = (rank-1 + p) % p;
right = (rank+1) % p;
...
MPI_Send(send_buffer1, size, MPI_CHAR, left, ...);
MPI_Recv(recv_buffer1, size, MPI_CHAR, right, ...);

MPI_Send(send_buffer2, size, MPI_CHAR, right, ...);
MPI_Recv(recv_buffer2, size, MPI_CHAR, left, ...);
...
```

(a) In the program, the processors are arranged in a logical ring and each processor should exchange its name with its neighbor to the left and its neighbor to the right. Assign a unique name to each MPI process and fill out the missing pieces of the program such that each process prints its own name as well as its neighbors' names.
(b) In the given program piece, the `MPI_Send()` and `MPI_Recv()` operations are arranged such that depending on the implementation a deadlock can occur. Describe how a deadlock may occur.
(c) Change the program such that no deadlock is possible by arranging the order of the `MPI_Send()` and `MPI_Recv()` operations appropriately.
(d) Change the program such that `MPI_Sendrecv()` is used to avoid deadlocks.
(e) Change the program such that `MPI_Isend()` and `MPI_Irecv()` are used.

Exercise 5.2 Consider the MPI program in Fig. 5.3 for the collection of distributed data block with point-to-point messages. The program assumes that all data blocks have the same size `blocksize`. Generalize the program such that each process can contribute a data block of a size different from the data blocks of the other processes. To do so, assume that each process has a local variable which specifies the size of its data block.

(*Hint*: First make the size of each data block available to each process in a pre-collection phase with a similar communication pattern as in Fig. 5.3 and then perform the actual collection of the data blocks.)

Exercise 5.3 Modify the program from the previous exercise for the collection of data blocks of different sizes such that no pre-collection phase is used. Instead, use `MPI_Get_count()` to determine the size of the data block received in each step. Compare the resulting execution time with the execution time of the program

from the previous exercise for different data block sizes and different numbers of processors. Which of the programs is faster?

Exercise 5.4 Consider the program `Gather_ring()` from Fig. 5.3. As described in the text, this program does not avoid deadlocks if the runtime system does not use internal system buffers. Change the program such that deadlocks are avoided in any case by arranging the order of the `MPI_Send()` and `MPI_Recv()` operations appropriately.

Exercise 5.5 The program in Fig. 5.3 arranges the processors logically in a ring to perform the collection. Modify the program such that the processors are logically arranged in a logical two-dimensional torus network. For simplicity, assume that all data blocks have the same size. Develop a mechanism with which each processor can determine its predecessor and successor in x and y directions. Perform the collection of the data blocks in two phases, the first phase with communication in x direction, the second phase with communication in y direction.

In both directions, communication in different rows or columns of the processor torus can be performed concurrently. For the communication in y direction, each process distributes all blocks that it has collected in the x direction phase. Use the normal blocking send and receive operations for the communication. Compare the resulting execution time with the execution time of the ring implementation from Fig. 5.3 for different data block sizes and different numbers of processors. Which of the programs is faster?

Exercise 5.6 Modify the program from the previous exercise such that non-blocking communication operations are used.

Exercise 5.7 Consider the parallel computation of a matrix–vector multiplication $A \cdot b$ using a distribution of the scalar products based on a rowwise distribution of A, see Fig. 3.10, p. 127 for a sketch of a parallel pseudo program. Transform this program into a running MPI program. Select the MPI communication operations for the multi-broadcast operations appropriately.

Exercise 5.8 Similar to the preceding exercise, consider a matrix–vector multiplication using a distribution of the linear combinations based on a columnwise distribution of the matrix. Transform the pseudo program from Fig. 3.12, p. 129 to a running MPI program. Use appropriate MPI operations for the single-accumulation and single-broadcast operations. Compare the execution time with the execution time of the MPI program from the preceding exercise for different sizes of the matrix.

Exercise 5.9 For a broadcast operation a root process sends the same data block to all other processes. Implement a broadcast operation by using point-to-point send and receive operations (`MPI_Send()` and `MPI_Recv()`) such that the same effect as `MPI_Bcast()` is obtained. For the processes, use a logical ring arrangement similar to Fig. 5.3.

Exercise 5.10 Modify the program from the previous exercise such that two other logical arrangements are used for the processes: a two-dimensional mesh and a

three-dimensional hypercube. Measure the execution time of the three different versions (ring, mesh, hypercube) for eight processors for different sizes of the data block and make a comparison by drawing a diagram. Use `MPI_Wtime()` for the timing.

Exercise 5.11 Consider the construction of conflict-free spanning trees in a d-dimensional hypercube network for the implementation of a multi-broadcast operation, see Sect. 4.3.2, p. 177, and Fig. 4.6. For $d = 3$, $d = 4$, and $d = 5$ write an MPI program with 8, 16, and 32 processes, respectively, that uses these spanning trees for a multi-broadcast operation.

(a) Implement the multi-broadcast by concurrent single-to-single transfers along the spanning trees and measure the resulting execution time for different message sizes.
(b) Implement the multi-broadcast by using multiple broadcast operations where each broadcast operation is implemented by single-to-single transfers along the usual spanning trees for hypercube networks as defined in p. 174, see Fig. 4.4. These spanning trees do not avoid conflicts in the network. Measure the resulting execution time for different message sizes and compare them with the execution times from (a).
(c) Compare the execution times from (a) and (b) with the execution time of an `MPI_Allgather()` operation to perform the same communication.

Exercise 5.12 For a global exchange operation, each process provides a potentially different block of data for each other process, see pp. 122 and 225 for a detailed explanation. Implement a global exchange operation by using point-to-point send and receive operations (`MPI_Send()` and `MPI_Recv()`) such that the same effect as `MPI_Alltoall()` is obtained. For the processes, use a logical ring arrangement similar to Fig. 5.3.

Exercise 5.13 Modify the program `Gather_ring()` from Fig. 5.3 such that synchronous send operations (`MPI_Send()` and `MPI_Recv()`) are used. Compare the resulting execution time with the execution time obtained for the standard send and receive operations from Fig. 5.3.

Exercise 5.14 Repeat the previous exercise with buffered send operations.

Exercise 5.15 Modify the program `Gather_ring()` from Fig. 5.3 such that the MPI operation `MPI_Test()` is used instead of `MPI_Wait()`. When a non-blocking receive operation is found by `MPI_Test()` to be completed, the process sends the received data block to the next process.

Exercise 5.16 Write an MPI program which implements a broadcast operation with `MPI_Send()` and `MPI_Recv()` operations. The program should use $n = 2^k$ processes which should logically be arranged as a hypercube network. Based on this arrangement the program should define a spanning tree in the network with root 0, see Fig. 3.8 and p. 123, and should use this spanning tree to transfer a message

stepwise from the root along the tree edges up to the leaves. Each node in the tree receives the message from its parent node and forwards it to its child nodes. Measure the resulting runtime for different message sizes up to 1 MB for different numbers of processors using `MPI_Wtime()` and compare the execution times with the execution times of `MPI_Bcast()` performing the same operation.

Exercise 5.17 The execution time of point-to-point communication operations between two processors can normally be described by a linear function of the form

$$t_{s2s}(m) = \tau + t_c \cdot m,$$

where m is the size of the message; τ is a startup time, which is independent of the message size; and t_c is the inverse of the network bandwidth. Verify this function by measuring the time for a ping-pong message transmission where process A sends a message to process B, and B sends the same message back to A. Use different message sizes and draw a diagram which shows the dependence of the communication time on the message size. Determine the size of τ and t_c on your parallel computer.

Exercise 5.18 Write an MPI program which arranges 24 processes in a (periodic) Cartesian grid structure of dimension $2 \times 3 \times 4$ using `MPI_Cart_create()`. Each process should determine and print the process rank of its two neighbors in x, y, and z directions.

For each of the three sub-grids in y-direction, a communicator should be defined. This communicator should then be used to determine the maximum rank of the processes in the sub-grid by using an appropriate `MPI_Reduce()` operation. This maximum rank should be printed out.

Exercise 5.19 Write an MPI program which arranges the MPI processes in a two-dimensional torus of size $\sqrt{p} \times \sqrt{p}$ where p is the number of processes. Each process exchanges its rank with its two neighbors in x and y dimensions. For the exchange, one-sided communication operations should be used. Implement three different schemes for the exchange with the following one-sided communication operations:

(a) global synchronization with `MPI_Win_fence()`;
(b) loose synchronization by using `MPI_Win_start()`, `MPI_Win_post()`, `MPI_Win_complete()`, and `MPI_Win_wait()`;
(c) lock synchronization with `MPI_Win_lock()` and `MPI_Win_unlock()`.

Test your program for $p = 16$ processors, i.e., for a 4×4 torus network.

Chapter 6
Thread Programming

Several parallel computing platforms, in particular multicore platforms, offer a shared address space. A natural programming model for these architectures is a thread model in which all threads have access to shared variables. These shared variables are then used for information and data exchange. To coordinate the access to shared variables, synchronization mechanisms have to be used to avoid race conditions in case of concurrent accesses. Basic synchronization mechanisms are lock synchronization and condition synchronization, see Sect. 3.7 for an overview.

In this chapter, we consider thread programming in more detail. In particular, we have a closer look at synchronization problems like deadlocks or priority inversion that might occur and present programming techniques to avoid such problems. Moreover, we show how basic synchronization mechanisms like lock synchronization or condition synchronization can be used to build more complex synchronization mechanisms like read/write locks. We also present a set of parallel patterns like task-based or pipelined processing that can be used to structure a parallel application. These issues are considered in the context of popular programming environments for thread-based programming to directly show the usage of the mechanisms in practice. The programming environments Pthreads, Java threads, and OpenMP are introduced in detail. For Java, we also give an overview of the package `java.util.concurrent` which provides many advanced synchronization mechanisms as well as a task-based execution environment. The goal of the chapter is to enable the reader to develop correct and efficient thread programs that can be used, for example, on multicore architectures.

6.1 Programming with Pthreads

POSIX threads (also called Pthreads) define a standard for the programming with threads, based on the programming language C. The threads of a process share a common address space. Thus, the global variables and dynamically generated data objects can be accessed by all threads of a process. In addition, each thread has a separate runtime stack which is used to control the functions activated and to store their local variables. These variables declared locally within the functions are *local*

T. Rauber, G. Rünger, *Parallel Programming*,
DOI 10.1007/978-3-642-04818-0_6,

data of the executing thread and cannot be accessed directly by other threads. Since the runtime stack of a thread is deleted after a thread is terminated, it is dangerous to pass a reference to a local variable in the runtime stack of a thread A to another thread B.

The data types, interface definitions, and macros of Pthreads are usually available via the header file <pthread.h>. This header file must therefore be included into a Pthreads program. The functions and data types of Pthreads are defined according to a naming convention. According to this convention, Pthreads functions are named in the form

```
pthread[_<object>]_<operation> (),
```

where <operation> describes the operation to be performed and the optional <object> describes the object to which this operation is applied. For example, pthread_mutex_init() is a function for the initialization of a mutex variable; thus, the <object> is mutex and the <operation> is init; we give a more detailed description later.

For functions which are involved in the manipulation of threads, the specification of <object> is omitted. For example, the function for the generation of a thread is pthread_create(). All Pthread functions yield a return value 0, if they are executed without failure. In case of a failure, an error code from <error.h> will be returned. Thus, this header file should also be included in the program. Pthread data types describe, similarly to MPI, opaque objects whose exact implementation is hidden from the programmer. Data types are named according to the syntax form

```
pthread_<object>_t,
```

where <object> specifies the specific data object. For example, a mutex variable is described by the data type pthread_mutex_t. If <object> is omitted, the data type pthread_t for threads results. The following table contains important Pthread data types which will be described in more detail later.

Pthread data types	Meaning
pthread_t	Thread ID
pthread_mutex_t	Mutex variable
pthread_cond_t	Condition variable
pthread_key_t	Access key
pthread_attr_t	Thread attributes object
pthread_mutexattr_t	Mutex attributes object
pthread_condattr_t	Condition variable attributes object
pthread_once_t	*One-time initialization* control context

For the execution of threads, we assume a two-step scheduling method according to Fig. 3.16 in Chap. 3, as this is the most general case. In this model, the programmer has to partition the program into a suitable number of user threads which can be executed concurrently with each other. The user threads are mapped

by the library scheduler to system threads which are then brought to execution on the processors of the computing system by the scheduler of the operating system. The programmer cannot control the scheduler of the operating system and has only little influence on the library scheduler. Thus, the programmer cannot directly perform the mapping of the user-level threads to the processors of the computing system, e.g., by a scheduling at program level. This facilitates program development, but also prevents an efficient mapping directly by the programmer according to his specific needs. It should be noted that there are operating system–specific extensions that allow thread execution to be bound to specific processors. But in most cases, the scheduling provided by the library and the operating system leads to good results and relieves the programmer from additional programming effort, thus providing more benefits than drawbacks.

In this section, we give an overview of the programming with Pthreads. Section 6.1.1 describes thread generation and management in Pthreads. Section 6.1.2 describes the lock mechanism for the synchronization of threads accessing shared variables. Sections 6.1.3 and 6.1.4 introduce Pthreads condition variables and an extended lock mechanism using condition variables, respectively. Sections 6.1.6, 6.1.7, and 6.1.8 describe the use of the basic synchronization techniques in the context of more advanced synchronization patterns, like task pools, pipelining, and client–server coordination. Section 6.1.9 discusses additional mechanisms for the control of threads, including scheduling strategies. We describe in Sect. 6.1.10 how the programmer can influence the scheduling controlled by the library. The phenomenon of *priority inversion* is then explained in Sect. 6.1.11 and finally thread-specific data is considered in Sect. 6.1.12. Only the most important mechanisms of the Pthreads standard are described; for a more detailed description, we refer to [25, 105, 117, 126, 143].

6.1.1 Creating and Merging Threads

When a Pthreads program is started, a single *main thread* is active, executing the `main()` function of the program. The main thread can generate more threads by calling the function

```
int pthread_create (pthread_t *thread,
                    const pthread_attr_t *attr,
                    void *(*start_routine)(void *),
                    void *arg).
```

The first argument is a pointer to an object of type `pthread_t` which is also referred to as *thread identifier* (TID); this TID is generated by `pthread_create()` and can later be used by other Pthreads functions to identify the generated thread. The second argument is a pointer to a previously allocated and initialized attribute object of type `pthread_attr_t`, defining the desired attributes of the generated thread. The argument value `NULL` causes the generation of a thread with default

attributes. If different attribute values are desired, an attribute data structure has to be created and initialized before calling `pthread_create()`; this mechanism is described in more detail in Sect. 6.1.9. The third argument specifies the function `start_routine()` which will be executed by the generated thread. The specified function should expect a single argument of type `void *` and should have a return value of the same type. The fourth argument is a pointer to the argument value with which the thread function `start_routine()` will be executed.

To execute a thread function with more than one argument, all arguments must be put into a single data structure; the address of this data structure can then be specified as argument of the thread function. If several threads are started by a parent thread using the same thread function but different argument values, *separate* data structures should be used for each of the threads to specify the arguments. This avoids situations where argument values are overwritten too early by the parent thread before they are read by the child threads or where different child threads manipulate the argument values in a common data structure concurrently.

A thread can determine its own thread identifier by calling the function

```
pthread_t pthread_self().
```

This function returns the thread ID of the calling thread. To compare the thread ID of two threads, the function

```
int pthread_equal (pthread_t t1, pthread_t t2)
```

can be used. This function returns the value 0 if `t1` and `t2` do not refer to the same thread. Otherwise, a non-zero value is returned. Since `pthread_t` is an opaque data structure, only `pthread_equal` should be used to compare thread IDs. The number of threads that can be generated by a process is typically limited by the system. The Pthreads standard determines that at least 64 threads can be generated by any process. But depending on the specific system used, this limit may be larger. For most systems, the maximum number of threads that can be started can be determined by calling

```
maxThreads = sysconf (_SC_THREAD_THREADS_MAX)
```

in the program. Knowing this limit, the program can avoid to start more than `maxThreads` threads. If the limit is reached, a call of the `pthread_create()` function returns the error value `EAGAIN`. A thread is terminated if its thread function terminates, e.g., by calling `return`. A thread can terminate itself explicitly by calling the function

```
void pthread_exit (void *valuep)
```

The argument `valuep` specifies the value that will be returned to another thread which waits for the termination of this thread using `pthread_join()`. When

a thread terminates its thread function, the function pthread_exit() is called *implicitly*, and the return value of the thread function is used as argument of this implicit call of pthread_exit(). After the call to pthread_exit(), the calling thread is terminated, and its runtime stack is freed and can be used by other threads. Therefore, the return value of the thread should not be a pointer to a local variable of the thread function or another function called by the thread function. These local variables are stored on the runtime stack and may not exist any longer after the termination of the thread. Moreover, the memory space of local variables can be reused by other threads, and it can usually not be determined when the memory space is overwritten, thereby destroying the original value of the local variable. Instead of a local variable, a global variable or a variable that has been dynamically allocated should be used.

A thread can wait for the termination of another thread by calling the function

```
int pthread_join (pthread_t thread, void **valuep).
```

The argument thread specifies the thread ID of the thread for which the calling thread waits to be terminated. The argument valuep specifies a memory address where the return value of this thread should be stored. The thread calling pthread_join() is blocked until the specified thread has terminated. Thus, pthread_join() provides a possibility for the *synchronization* of threads. After the thread with TID thread has terminated, its return value is stored at the specified memory address. If several threads wait for the termination of the same thread, using pthread_join(), all waiting threads are blocked until the specified thread has terminated. But only one of the waiting threads successfully stores the return value. For all other waiting threads, the return value of pthread_join() is the error value ESRCH. The runtime system of the Pthreads library allocates for each thread an internal data structure to store information and data needed to control the execution of the thread. This internal data structure is preserved by the runtime system also after the termination of the thread to ensure that another thread can later successfully access the return value of the terminated thread using pthread_join().

After the call to pthread_join(), the internal data structure of the terminated thread is released and can no longer be accessed. If there is no pthread_join() for a specific thread, its internal data structure is not released after its termination and occupies memory space until the complete process is terminated. This can be a problem for large programs with many thread creations and terminations without corresponding calls to pthread_join(). The preservation of the internal data structure of a thread after its termination can be avoided by calling the function

```
int pthread_detach (pthread_t thread).
```

This function notifies the runtime system that the internal data structure of the thread with TID thread can be detached as soon as the thread has terminated. A thread may detach itself, and any thread may detach any other thread. After a thread has

```
#include <pthread.h>

typedef struct {
  int size, row, column;
  double (*MA)[8], (*MB)[8], (*MC)[8];
} matrix_type_t;

void *thread_mult (void *w) {
  matrix_type_t *work = (matrix_type_t *) w;
  int i, row = work->row, column = work->column;
  work -> MC[row][column] = 0;
  for (i=0; i < work->size; i++)
    work->MC[row][column] += work->MA[row][i] * work->MB[i][column];
  return NULL;
}

int main() {
  int row, column, size = 8, i;
  double MA[8][8], MB[8][8], MC[8][8];
  matrix_type_t *work;
  pthread_t thread[8*8];
  for (row=0; row<size; row++)
    for (column=0; column<size; column++) {
      work = (matrix_type_t *) malloc (sizeof (matrix_type_t));
      work->size = size;
      work->row = row;
      work->column = column;
      work->MA = MA; work->MB = MB; work->MC = MC;
      pthread_create (&(thread[column + row*8]), NULL,
                      thread_mult, (void *) work);
    }

  for (i=0; i<size*size; i++)
    pthread_join (thread[i], NULL);
}
```

Fig. 6.1 Pthreads program for the multiplication of two matrices `MA` and `MB`. A separate thread is created for each element of the output matrix `MC`. A separate data structure `work` is provided for each of the threads created

been set into a detached state, calling `pthread_join()` for this thread returns the error value `EINVAL`.

Example We give a first example for a Pthreads program; Fig. 6.1 shows a program fragment for the multiplication of two matrices, see also [126]. The matrices `MA` and `MB` to be multiplied have a fixed size of eight rows and eight columns. For each of the elements of the result matrix `MC`, a separate thread is created. The IDs of these threads are stored in the array `thread`. Each thread obtains a separate data structure of type `matrix_type_t` which contains pointers to the input matrices `MA` and `MB`, the output matrix `MC`, and the row and column position of the entry of `MC` to be computed by the corresponding thread. Each thread executes the same thread function `thread_mult()` which computes the scalar product of one row of `MA` and one column of `MB`. After creating a new thread for each of the 64 elements

of MC to be computed, the main thread waits for the termination of each of these threads using pthread_join(). The program in Fig. 6.1 creates 64 threads which is exactly the limit defined by the Pthreads standard for the number of threads that must be supported by each implementation of the standard. Thus, the given program works correctly. But it is not scalable in the sense that it can be extended to the multiplication of matrices of any size. Since a separate thread is created for each element of the output matrix, it can be expected that the upper limit for the number of threads that can be generated will be reached even for matrices of moderate size. Therefore, the program should be re-written when using larger matrices such that a fixed number of threads is used and each thread computes a block of entries of the output matrix; the size of the blocks increases with the size of the matrices. □

6.1.2 Thread Coordination with Pthreads

The threads of a process share a common address space. Therefore, they can concurrently access shared variables. To avoid race conditions, these concurrent accesses must be coordinated. To perform such coordinations, Pthreads provide *mutex variables* and *condition variables*.

6.1.2.1 Mutex Variables

In Pthreads, a **mutex variable** denotes a data structure of the predefined opaque type pthread_mutex_t. Such a mutex variable can be used to ensure *mutual exclusion* when accessing common data, i.e., it can be ensured that only one thread at a time has exclusive access to a common data structure, all other threads have to wait. A mutex variable can be in one of two states: *locked* and *unlocked*. To ensure mutual exclusion when accessing a common data structure, a separate mutex variable is assigned to the data structure. All accessing threads must behave as follows: *Before* an access to the common data structure, the accessing thread locks the corresponding mutex variable using a specific Pthreads function. When this is successful, the thread is the *owner* of the mutex variable. *After* each access to the common data structure, the accessing thread unlocks the corresponding mutex variable. After the unlocking, it is no longer the owner of the mutex variable, and another thread can become the owner and is allowed to access the data structure.

When a thread A tries to lock a mutex variable that is already owned by another thread B, thread A is blocked until thread B unlocks the mutex variable. The Pthreads runtime system ensures that only one thread at a time is the owner of a specific mutex variable. Thus, a conflicting manipulation of a common data structure is avoided if each thread uses the described behavior. But if a thread accesses the data structure without locking the mutex variable before, mutual exclusion is no longer guaranteed.

The assignment of mutex variables to data structures is done implicitly by the programmer by protecting accesses to the data structure with locking and unlocking

operations of a specific mutex variable. There is no explicit assignment of mutex variables to data structures. The programmer can improve the readability of Pthreads programs by grouping a common data structure and the protecting mutex variable into a new structure.

In Pthreads, mutex variables have the predefined type `pthread_mutex_t`. Like normal variables, they can be statically declared or dynamically generated. Before a mutex variable can be used, it must be initialized. For a mutex variable `mutex` that is allocated statically, this can be done by

```
mutex = PTHREAD_MUTEX_INITIALIZER
```

where `PTHREAD_MUTEX_INITIALIZER` is a predefined macro. For arbitrary mutex variables (statically allocated or dynamically generated), an initialization can be performed dynamically by calling the function

```
int pthread_mutex_init (pthread_mutex_t *mutex,
                        const pthread_mutexattr_t *attr).
```

For `attr = NULL`, a mutex variable with default properties results. The properties of mutex variables can be influenced by using different attribute values, see Sect. 6.1.9. If a mutex variable that has been initialized dynamically is no longer needed, it can be destroyed by calling the function

```
int pthread_mutex_destroy (pthread_mutex_t *mutex).
```

A mutex variable should only be destroyed if none of the threads is waiting for the mutex variable to become owner and if there is currently no owner of the mutex variable. A mutex variable that has been destroyed can later be re-used after a new initialization. A thread can lock a mutex variable `mutex` by calling the function

```
int pthread_mutex_lock (pthread_mutex_t *mutex).
```

If another thread B is owner of the mutex variable `mutex` when a thread A issues the call of `pthread_mutex_lock()`, then thread A is blocked until thread B unlocks `mutex`. When several threads $T_1, \ldots, T_n$ try to lock a mutex variable which is owned by another thread, all threads $T_1, \ldots, T_n$ are blocked and are stored in a waiting queue for this mutex variable. When the owner releases the mutex variable, one of the blocked threads in the waiting queue is unblocked and becomes the new owner of the mutex variable. Which one of the waiting threads is unblocked may depend on their priorities and the scheduling strategies used, see Sect. 6.1.9 for more information. The order in which waiting threads become owner of a mutex variable is not defined in the Pthreads standard and may depend on the specific Pthreads library used.

A thread should not try to lock a mutex variable when it is already the owner. Depending on the specific runtime system, this may lead to an error return value EDEADLK or may even cause a self-deadlock. A thread which is owner of a mutex variable `mutex` can unlock `mutex` by calling the function

```
int pthread_mutex_unlock (pthread_mutex_t *mutex).
```

After this call, `mutex` is in the state *unlocked*. If there is no other thread waiting for `mutex`, there is no owner of `mutex` after this call. If there are threads waiting for `mutex`, one of these threads is woken up and becomes the new owner of `mutex`. In some situations, it is useful that a thread can check without blocking whether a mutex variable is owned by another thread. This can be achieved by calling the function

```
int pthread_mutex_trylock (pthread_mutex_t *mutex).
```

If the specified mutex variable is currently not held by another thread, the calling thread becomes the owner of the mutex variable. This is the same behavior as for `pthread_mutex_lock()`. But different from `pthread_mutex_lock()`, the calling thread is *not blocked* if another thread already holds the mutex variable. Instead, the call returns with error return value `EBUSY` without blocking. The calling thread can then perform other computations and can later retry to lock the mutex variable. The calling thread can also repeatedly try to lock the mutex variable until it is successful (*spinlock*).

Example Figure 6.2 shows a simple program fragment to illustrate the use of mutex variables to ensure mutual exclusion when concurrently accessing a common data structure, see also [126]. In the example, the common data structure is a linked list. The nodes of the list have type `node_t`. The complete list is protected by a single mutex variable. To indicate this, the pointer to the first element of the list (`first`) is combined with the mutex variable (`mutex`) into a data structure of type `list_t`. The linked list will be kept sorted according to increasing values of the node entry `index`. The function `list_insert()` inserts a new element into the list while keeping the sorting. Before the first call to `list_insert()`, the list must be initialized by calling `list_init()`, e.g., in the main thread. This call also initializes the mutex variable. In `list_insert()`, the executing thread first locks the mutex variable of the list before performing the actual insertion. After the insertion, the mutex variable is released again using `pthread_mutex_unlock()`. This procedure ensures that it is not possible for different threads to insert new elements at the same time. Hence, the list operations are *sequentialized*. The function `list_insert()` is a *thread-safe* function, since a program can use this function without performing additional synchronization.

In general, a (library) function is thread-safe if it can be called by different threads concurrently, without performing additional operations to avoid race conditions. □

```
typedef struct node {
  int index;
  void *data;
  struct node *next;
} node_t;

typedef struct list {
  node_t *first;
  pthread_mutex_t mutex;
} list_t;

void list_init (list_t *listp)
{
  listp->first = NULL;
  pthread_mutex_init (&(listp->mutex), NULL);
}

void list_insert (int newindex, void *newdata, list_t *listp)
{
  node_t *current, *previous, *new;
  int found = FALSE;

  pthread_mutex_lock (&(listp->mutex));
  for (current = previous = listp->first; current != NULL;
       previous = current, current = current->next)
  {
    if (current->index == newindex) {
      found = TRUE; break;
    }
    else
      if (current->index > newindex) break;
  }
  if (!found) {
    new = (node_t *) malloc (sizeof (node_t));
    new->index = newindex;
    new->data = newdata;
    new->next = current;
    if (current == listp->first) lstp->first = new;
    else previous->next = new;
  }
  pthread_mutex_unlock (&(lstp->mutex));
}
```

Fig. 6.2 Pthreads implementation of a linked list. The function `list_insert()` can be called by different threads concurrently which insert new elements into the list. In the form presented, `list_insert()` cannot be used as the start function of a thread, since the function has more than one argument. To be used as start function, the arguments of `list_insert()` have to be put into a new data structure which is then passed as argument. The original arguments could then be extracted from this data structure at the beginning of `list_insert()`

In Fig. 6.2, a single mutex variable is used to control the complete list. This results in a *coarse-grain* lock granularity. Only a single insert operation can happen at a time, independently of the length of the list. An alternative could be to partition the list into fixed-size areas and protect each area with a mutex variable or even to protect each single element of the list with a separate mutex variable. In this case, the granularity would be **fine-grained**, and several threads could access different parts of the list concurrently. But this also requires a substantial re-organization of the synchronization, possibly leading to a larger overhead.

6.1.2.2 Mutex Variables and Deadlocks

When multiple threads work with different data structures each of which is protected by a separate mutex variable, caution has to be taken to avoid deadlocks. A deadlock may occur if the threads use a different order for locking the mutex variables. This can be seen for two threads T_1 and T_2 and two mutex variables *ma* and *mb* as follows:

- thread T_1 first locks *ma* and then *mb*;
- thread T_2 first locks *mb* and then *ma*.

If T_1 is interrupted by the scheduler of the runtime system after locking *ma* such that T_2 is able to successfully lock *mb*, a deadlock occurs:
T_2 will be blocked when it is trying to lock *ma*, since *ma* is already locked by T_1; similarly, T_1 will be blocked when it is trying to lock *mb* after it has been woken up again, since *mb* has already been locked by T_2. In effect, both threads are blocked forever and are mutually waiting for each other. The occurrence of deadlocks can be avoided by using a *fixed locking order* for all threads or by employing a *backoff strategy*.

When using a **fixed locking order**, each thread locks the critical mutex variables always in the same predefined order. Using this approach for the example above, thread T_2 must lock the two mutex variables *ma* and *mb* in the same order as T_1, e.g., both threads must first lock *ma* and then *mb*. The deadlock described above cannot occur now, since T_2 cannot lock *mb* if *ma* has previously been locked by T_1. To lock *mb*, T_2 must first lock *ma*. If *ma* has already been locked by T_1, T_2 will be blocked when trying to lock *ma* and, hence, cannot lock *mb*. The specific locking order used can in principle be arbitrarily selected, but to avoid deadlocks it is important that the order selected is used throughout the entire program. If this does not conform to the program structure, a backoff strategy should be used.

When using a **backoff strategy**, each participating thread can lock the mutex variables in its individual order, and it is not necessary to use the same predefined order for each thread. But a thread must back off when its attempt to lock a mutex variable fails. In this case, the thread must release all mutex variables that it has previously locked successfully. After the backoff, the thread starts the entire lock procedure from the beginning by trying to lock the first mutex variable again. To implement a backoff strategy, each thread uses `pthread_mutex_lock()` to lock its first mutex variable and `pthread_mutex_trylock()` to lock the remaining

mutex variables needed. If pthread_mutex_trylock() returns EBUSY, this means that this mutex variable is already locked by another thread. In this case, the calling thread releases all mutex variables that it has previously locked successfully using pthread_mutex_unlock().

Example Backoff strategy (see Figs. 6.3 and 6.4):
The use of a backoff strategy is demonstrated in Fig. 6.3 for two threads f and b which lock three mutex variables m[0], m[1], and m[2] in different orders, see [25]. The thread f (forward) locks the mutex variables in the order m[0], m[1], and m[2] by calling the function lock_forward(). The thread b (backward) locks the mutex variables in the opposite order m[2], m[1], and m[0] by calling the function lock_backward(), see Fig. 6.4. Both threads repeat the locking 10 times. The main program in Fig. 6.3 uses two control variables backoff and yield_flag which are read in as arguments. The control variable backoff determines whether a backoff strategy is used (value 1) or not (value 0). For backoff = 1, no deadlock occurs when running the program because of the backoff strategy. For backoff = 0, a deadlock occurs in most cases, in particular if f succeeds in locking m[0] and b succeeds in locking m[2].

But depending on the specific scheduling situation concerning f and b, no deadlock may occur even if no backoff strategy is used. This happens when both threads succeed in locking all three mutex variables, before the other thread is executed. To illustrate this dependence of deadlock occurrence from the specific scheduling situation, the example in Figs. 6.3 and 6.4 contains a mechanism to influence the scheduling of f and b. This mechanism is activated by using the control variable yield_flag. For yield_flag = 0, each thread tries to lock the mutex variables without interruption. This is the behavior described so far. For yield_flag = 1, each thread calls sched_yield() after having locked a mutex variable, thus transferring control to another thread with the same priority. Therefore, the other

```
#include <pthread.h>
#include <sched.h>
#include <stdlib.h>
#include <stdio.h>
pthread_mutex_t m[3] = {
  PTHREAD_MUTEX_INITIALIZER,
  PTHREAD_MUTEX_INITIALIZER,
  PTHREAD_MUTEX_INITIALIZER };
int backoff = 1; // == 1: with backoff strategy
int yield_flag = 0; // > 0: use sched_yield, < 0: sleep
int main(int argc, char *argv[]) {
  pthread_t f, b;
  if (argc > 1) backoff = atoi(argv[1]);
  if (argc > 2) yield_flag = atoi(argv[2]);
  pthread_create(&f, NULL, lock_forward, NULL);
  pthread_create(&b, NULL, lock_backward, NULL);
  pthread_exit(NULL); // both threads continue execution
}
```

Fig. 6.3 Control program to illustrate the use of a backoff strategy

```
void *lock_forward(void *arg) {
  int iterate, i, status;
  for (iterate = 0; iterate < 10; iterate++) {
    for (i = 0; i < 3; i++) { // lock order forward
       if (i == 0 || !backoff)
         status = pthread_mutex_lock(&m[i]);
       else status = pthread_mutex_trylock(&m[i]);
       if (status == EBUSY)
         for (--i; i >= 0; i--) pthread_mutex_unlock(&m[i]);
       else printf("forward locker got mutex %d\n", i);
       if (yield_flag) {
         if (yield_flag > 0) sched_yield(); // switch threads
         else sleep(1); // block executing thread
       }
    }
    for (i = 2; i >= 0; i--)
       pthread_mutex_unlock(&m[i]);
    sched_yield(); // new trial with potentially different order
  }
}
void *lock_backward(void *arg) {
  int iterate, i, status;
  for (iterate = 0; iterate < 10; iterate++) {
    for (i = 2; i >= 0; i--) { // lock order backward
       if (i == 2 || !backoff)
         status = pthread_mutex_lock(&m[i]);
       else status = pthread_mutex_trylock(&m[i]);
       if (status == EBUSY)
         for (++i; i < 3; i++) pthread_mutex_unlock(&m[i]);
       else printf("backward locker got mutex %d\n", i);
       if (yield_flag)
         if (yield_flag > 0) sched_yield();
         else sleep(1);
    }
    for (i = 0; i < 3; i++)
       pthread_mutex_unlock(&m[i]);
    sched_yield();
  }
}
```

Fig. 6.4 Functions `lock_forward` and `lock_backward` to lock mutex variables in opposite directions

thread has a chance to lock a mutex variable. For `yield_flag = -1`, each thread calls `sleep(1)` after having locked a mutex variable, thus waiting for 1 s. In this time, the other thread can run and has a chance to lock another mutex variable. In both cases, a deadlock will likely occur if no backoff strategy is used.

Calling `pthread_exit()` in the main thread causes the termination of the main thread, but not of the entire process. Instead, using a normal `return` would terminate the entire process, including the threads `f` and `b`. □

Compared to a fixed locking order, the use of a backoff strategy typically leads to larger execution times, since threads have to back off when they do not succeed in locking a mutex variable. In this case, the locking of the mutex variables has to be started from the beginning.

But using a backoff strategy leads to an increased flexibility, since no fixed locking order has to be ensured. Both techniques can also be used in combination

by using a fixed locking order in code regions where this is not a problem and using a backoff strategy where the additional flexibility is beneficial.

6.1.3 Condition Variables

Mutex variables are typically used to ensure mutual exclusion when accessing global data structures concurrently. But mutex variables can also be used to wait for the occurrence of a specific condition which depends on the state of a global data structure and which has to be fulfilled before a certain operation can be applied. An example might be a shared buffer from which a consumer thread can remove entries only if the buffer is not empty. To apply this mechanism, the shared data structure is protected by one or several mutex variables, depending on the specific situation. To check whether the condition is fulfilled, the executing thread locks the mutex variable(s) and then evaluates the condition. If the condition is fulfilled, the intended operation can be performed. Otherwise, the mutex variable(s) are released again and the thread repeats this procedure again at a later time. This method has the drawback that the thread which is waiting for the condition to be fulfilled may have to repeat the evaluation of the condition quite often before the condition becomes true. This consumes execution time (*active waiting*), in particular because the mutex variable(s) have to be locked before the condition can be evaluated. To enable a more efficient method for waiting for a condition, Pthreads provide condition variables.

A **condition variable** is an opaque data structure which enables a thread to wait for the occurrence of an arbitrary condition without active waiting. Instead, a signaling mechanism is provided which blocks the executing thread during the waiting time, so that it does not consume CPU time. The waiting thread is woken up again as soon as the condition is fulfilled. To use this mechanism, the executing thread must define a condition variable and a mutex variable. The mutex variable is used to protect the evaluation of the specific condition which is waiting to be fulfilled. The use of the mutex variable is necessary, since the evaluation of a condition usually requires to access shared data which may be modified by other threads concurrently.

A condition variable has type `pthread_cond_t`. After the declaration or the dynamic generation of a condition variable, it must be initialized before it can be used. This can be done dynamically by calling the function

```
int pthread_cond_init (pthread_cond_t *cond,
                       const pthread_condattr_t *attr)
```

where `cond` is the address of the condition variable to be initialized and `attr` is the address of an attribute data structure for condition variables. Using `attr=NULL` leads to an initialization with the default attributes. For a condition variable `cond` that has been declared statically, the initialization can also be obtained by using the `PTHREAD_COND_INITIALIZER` initialization macro. This can also be done directly with the declaration

```
pthread_cond_t cond = PTHREAD_COND_INITIALIZER.
```

The initialization macro cannot be used for condition variables that have been generated dynamically using, e.g., `malloc()`. A condition variable `cond` that has been initialized with `pthread_cond_init()` can be destroyed by calling the function

```
int pthread_cond_destroy (pthread_cond_t *cond)
```

if it is no longer needed. In this case, the runtime system can free the information stored for this condition variable. Condition variables that have been initialized statically with the initialization macro do not need to be destroyed.

Each condition variable must be uniquely associated with a specific mutex variable. All threads which wait for a condition variable at the same time must use the same associated mutex variable. It is not allowed that different threads associate different mutex variables with a condition variable at the same time. But a mutex variable can be associated with different condition variables. A condition variable should only be used for a single condition to avoid deadlocks or race conditions [25]. A thread must first lock the associated mutex variable `mutex` with `pthread_mutex_lock()` before it can wait for a specific condition to be fulfilled using the function

```
int pthread_cond_wait (pthread_cond_t *cond,
                       pthread_mutex_t *mutex)
```

where `cond` is the condition variable used and `mutex` is the associated mutex variable. The condition is typically embedded into a surrounding control statement. A standard usage pattern is

```
pthread_mutex_lock (&mutex);
while (!condition())
  pthread_cond_wait (&cond, &mutex);
compute_something();
pthread_mutex_unlock (&mutex);
```

The evaluation of the condition and the call of `pthread_cond_wait()` are protected by the mutex variable `mutex` to ensure that the condition does not change between the evaluation and the call of `pthread_cond_wait()`, e.g., because another thread changes the value of a variable that is used within the condition. Therefore, each thread must use this mutex variable `mutex` to protect the manipulation of each variable that is used within the condition. Two cases can occur for this usage pattern for condition variables:

- If the specified condition is fulfilled when executing the code segment from above, the function `pthread_cond_wait()` is **not** called. The executing

thread releases the mutex variable and proceeds with the execution of the succeeding program part.

- If the specified condition is not fulfilled, `pthread_cond_wait()` is called. This call has the effect that the specified mutex variable `mutex` is implicitly released and that the executing thread is blocked, waiting for the condition variable until another thread sends a signal using `pthread_cond_signal()` to notify the blocked thread that the condition may now be fulfilled. When the blocked thread is woken up again in this way, it implicitly tries to lock the mutex variable `mutex` again. If this is owned by another thread, the woken-up thread is blocked again, now waiting for the mutex variable to be released. As soon as the thread becomes the owner of the mutex variable `mutex`, it continues the execution of the program. In the context of the usage pattern from above, this results in a new evaluation of the condition because of the `while` loop.

In a Pthreads program, it should be ensured that a thread which is waiting for a condition variable is woken up only if the specified condition is fulfilled. Nevertheless, it is useful to evaluate the condition again after the wake up because there are other threads working concurrently. One of these threads might become the owner of the mutex variable before the woken-up thread. Thus the woken-up thread is blocked again. During the blocking time, the owner of the mutex variable may modify common data such that the condition is no longer fulfilled. Thus, from the perspective of the executing thread, the state of the condition may change in the time interval between being woken up and becoming owner of the associated mutex variable. Therefore, the thread must again evaluate the condition to be sure that it is still fulfilled. If the condition is fulfilled, it cannot change before the executing thread calls `pthread_mutex_unlock()` or `pthread_cond_wait()` for the same condition variable, since each thread must be the owner of the associated mutex variable to modify a variable used in the evaluation of the condition.

Pthreads provide two functions to wake up (*signal*) a thread waiting on a condition variable:

```
int pthread_cond_signal (pthread_cond_t *cond)
int pthread_cond_broadcast (pthread_cond_t *cond).
```

A call of `pthread_cond_signal()` wakes up a *single* thread waiting on the condition variable `cond`. A call of this function has no effect, if there are no threads waiting for `cond`. If there are several threads waiting for `cond`, one of them is selected to be woken up. For the selection, the priorities of the waiting threads and the scheduling method used are taken into account. A call of `pthread_cond_broadcast()` wakes up *all* threads waiting on the condition variable `cond`. If several threads are woken up, only one of them can become owner of the associated mutex variable. All other threads that have been woken up are blocked on the mutex variable.

The functions `pthread_cond_signal()` and `pthread_cond_broadcast()` should only be called if the condition associated with `cond` is fulfilled. Thus, before calling one of these functions, a thread should evaluate the condition. To

do so safely, it must first lock the mutex variable associated with the condition variable to ensure a consistent evaluation of the condition. The actual call of `pthread_cond_signal()` or `pthread_cond_broadcast()` does not need to be protected by the mutex variable. Issuing a call without protection by the mutex variable has the drawback that another thread may become the owner of the mutex variable when it has been released after the evaluation of the condition, but before the signaling call. In this situation, the new owner thread can modify shared variables such that the condition is no longer fulfilled. This does not lead to an error, since the woken-up thread will again evaluate the condition. The advantage of not protecting the call of `pthread_cond_signal()` or `pthread_cond_broadcast()` by the mutex variable is the chance that the mutex variable may not have an owner when the waiting thread is woken up. Thus, there is a chance that this thread becomes the owner of the mutex variable without waiting. If mutex protection is used, the signaling thread is the owner of the mutex variable when the signal arrives, so the woken-up thread must block on the mutex variable immediately after being woken up.

To wait for a condition, Pthreads also provide the function

```
int pthread_cond_timedwait(pthread_cond_t *cond,
                           pthread_mutex_t *mutex,
                           const struct timespec *time).
```

The difference from `pthread_cond_wait()` is that the blocking on the condition variable `cond` is ended with return value `ETIMEDOUT` after the specified time interval `time` has elapsed. This maximum waiting time is specified using type

```
struct timespec {
  time_t tv_sec;
  long tv_nsec;
}
```

where `tv_sec` specifies the number of seconds and `tv_nsec` specifies the number of additional nanoseconds. The `time` parameter of `pthread_cond_timedwait()` specifies an absolute clock time rather than a time interval. A typical use may look as follows:

```
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t c = PTHREAD_COND_INITIALIZER;
struct timespec time;
pthread_mutex_lock (&m);
time.tv_sec = time (NULL) + 10;
time.tv_nsec = 0;
while (!Bedingung)
  if (pthread_cond_timedwait (&c, &m, &time) == ETIMEDOUT)
    timed_out_work();
pthread_mutex_unlock (&m);
```

In this example, the executing thread waits at most 10 s for the condition to be fulfilled. The function `time()` from `<time.h>` is used to define `time.tv_sec`. The call `time(NULL)` yields the absolute time in seconds elapsed since Jan 1, 1970. If no signal arrives after 10 s, the function `timed_out_work()` is called before the condition is evaluated again.

6.1.4 Extended Lock Mechanism

Condition variables can be used to implement more complex synchronization mechanisms that are not directly supported by Pthreads. In the following, we consider a *read/write lock* mechanism as an example for an extension of the standard lock mechanism provided by normal mutex variables. If we use a normal mutex variable to protect a shared data structure, only one thread at a time can access (read or write) the shared data structure. The following user-defined read/write locks extend this mechanism by allowing an arbitrary number of reading threads at a time. But only one thread at a time is allowed to write to the data structure. In the following, we describe a simple implementation of this extension, see also [126]. For more complex and more efficient implementations, we refer to [25, 105].

For the implementation of read/write locks, we define read/write lock variables (r/w lock variables) by combining a mutex variable and a condition variable as follows:

```
typedef struct rw_lock {
  int num_r, num_w;
  pthread_mutex_t mutex;
  pthread_cond_t cond;
} rw_lock_t;
```

Here, `num_r` specifies the current number of read permits, and `num_w` specifies the current number of write permits; `num_w` should have a maximum value of 1. The mutex variable `mutex` is used to protect the access to `num_r` and `num_w`. The condition variable `cond` coordinates the access to the r/w lock variable.

Figure 6.5 shows the functions that can be used to implement the read/write lock mechanism. The function `rw_lock_init()` initializes a read/write lock variable. The function `rw_lock_rlock()` requests a read permit to the common data structure. The read permit is granted only if there is no other thread that currently has a write permit. Otherwise the calling thread is blocked until the write permit is returned. The function `rw_lock_wlock()` requests a write permit to the common data structure. The write permit is granted only if there is no other thread that currently has a read or write permit.

The function `rw_lock_runlock()` is used to return a read permit. This may cause the number of threads with a read permit to decrease to zero. In this case, a

Fig. 6.5 Function for the control of read/write lock variables

```
int rw_lock_init (rw_lock_t *rwl) {
  rwl->num_r = rwl->num_w = 0;
  pthread_mutex_init (&(rwl->mutex),NULL);
  pthread_cond_init (&(rwl->cond),NULL);
  return 0;
}

int rw_lock_rlock (rw_lock_t *rwl) {
  pthread_mutex_lock (&(rwl->mutex));
  while (rwl->num_w > 0)
    pthread_cond_wait (&(rwl->cond), &(rwl->mutex));
  rwl->num_r ++;
  pthread_mutex_unlock (&(rwl->mutex));
  return 0;
}

int rw_lock_wlock (rw_lock_t *rwl) {
  pthread_mutex_lock (&(rwl->mutex));
  while ((rwl->num_w > 0) || (rwl->num_r > 0))
    pthread_cond_wait (&(rwl->cond), &(rwl->mutex));
  rwl->num_w ++;
  pthread_mutex_unlock (&(rwl->mutex));
  return 0;
}

int rw_lock_runlock (rw_lock_t *rwl) {
  pthread_mutex_lock (&(rwl->mutex));
  rwl->num_r --;
  if (rwl->num_r == 0) pthread_cond_signal (&(rwl->cond));
  pthread_mutex_unlock (&(rwl->mutex));
  return 0;
}

int rw_lock_wunlock (rw_lock_t *rwl) {
  pthread_mutex_lock (&(rwl->mutex));
  rwl->num_w --;
  pthread_cond_broadcast (&(rwl->cond));
  pthread_mutex_unlock (&(rwl->mutex));
  return 0;
}
```

thread which is waiting for a write permit is woken up by `pthread_cond_signal()`. The function `rw_lock_wunlock()` is used to return a write permit. Since only one thread with a write permit is allowed, there cannot be a thread with a write permit after this operation. Therefore, all threads waiting for a read or write permit can be woken up using `pthread_cond_broadcast()`.

The implementation sketched in Fig. 6.5 favors read requests over write requests: If a thread A has a read permit and a thread B waits for a write permit, then other threads will obtain a read permit without waiting, even if they put their read request long after B has put its write request. Thread B will get a write permit only if there are no other threads requesting a read permit. Depending on the intended usage, it might also be useful to give write requests priority over read requests to keep a data structure up to date. An implementation for this is given in [25].

The r/w lock mechanism can be used for the implementation of a shared linked list, see Fig. 6.2, by replacing the mutex variable `mutex` by a r/w lock variable. In the `list_insert()` function, the list access will then be protected by `rw_lock_wlock()` and `rw_lock_wunlock()`. A function to search for a specific entry in the list could use `rw_lock_rlock()` and `tw_lock_runlock()`, since no entry of the list will be modified when searching.

6.1.5 One-Time Initialization

In some situations, it is useful to perform an operation only once, no matter how many threads are involved. This is useful for initialization operations or opening a file. If several threads are involved, it sometimes cannot be determined in advance which of the threads is first ready to perform an operation. A one-time initialization can be achieved using a boolean variable initialized to 0 and protected by a mutex variable. The first thread arriving at the critical operation sets the boolean variable to 1, protected by the mutex variable, and then performs the one-time operation. If a thread arriving at the critical operation finds that the boolean variable has value 1, it does not perform the operation. Pthreads provide another solution for one-time operations by using a control variable of the predefined type `pthread_once_t`. This control variable must be statically initialized using the initialization macro `PTHREAD_ONCE_INIT`:

```
pthread_once_t once_control = PTHREAD_ONCE_INIT
```

The code to perform the one-time operation must be put into a separate function without parameter. We call this function `once_routine()` in the following. The one-time operation is then performed by calling the function

```
pthread_once (pthread_once_t *once_control,
              void (*once_routine)(void)).
```

This function can be called by several threads. If the execution of `once_routine()` has already been completed, then control is directly returned to the calling thread. If the execution of `once_routine()` has not yet been started, `once_routine()` is executed by the calling thread. If the execution of the function `once_routine()` has been started by another thread, but is not finished yet, then the thread executing `pthread_once()` waits until the other thread has finished its execution of `once_routine()`.

6.1.6 Implementation of a Task Pool

A thread program usually has to perform several operations or tasks. A simple structure results if each task is put into a separate function which is then called by a

separate thread which executes exactly this function and then terminates. Depending on the granularity of the tasks, this may lead to the generation and termination of a large number of threads, causing a significant overhead. For many applications, a more efficient implementation can be obtained by using a **task pool** (also called *work crew*). The idea is to use a specific data structure (task pool) to store the tasks that are ready for execution. For task execution, a fixed number of threads is used which are generated by the main thread at program start and exist until the program terminates. The threads access the task pool to retrieve tasks for execution. During the execution of a task, new tasks may be generated which are then inserted into the task pool. The execution of the parallel program is terminated if the task pool is empty and each thread has finished the execution of its task.

The advantage of this execution scheme is that a fixed number of threads is used, no matter how many tasks are generated. This keeps the overhead for thread management small, independent of the number of tasks. Moreover, tasks can be generated dynamically, thus enabling the realization of adaptive and irregular applications. In the following, we describe a simple implementation of a task pool, see also [126]. More advanced implementations are described in [25, 105].

Figure 6.6 presents the data structure that can be used for the task pool and a function for the initialization of the task pool. The data type `work_t` represents a single task. It contains a reference `routine` to the function containing the code of the task and the argument `arg` of this function. The tasks are organized as a linked list, and `next` is a pointer to the next task element. The data type `tpool_t` represents the actual task pool. It contains pointers `head` and `tail` to the first and last elements of the task list, respectively. The entry `num_threads` specifies the number of threads used for execution of the tasks. The array `threads` contains the reference to the thread IDs of these threads. The entries `max_size` and `current_size` specify the maximum and current number of tasks contained in the task pool.

The mutex variable `lock` is used to ensure mutual exclusion when accessing the task pool. If a thread attempts to retrieve a task from an empty task pool, it is blocked on the condition variable `not_empty`. If a thread inserts a task into an empty task pool, it wakes up a thread that is blocked on `not_empty`. If a thread attempts to insert a task into a full task pool, it is blocked on the condition variable `not_full`. If a thread retrieves a task from a full task pool, it wakes up a thread that is blocked on `not_full`.

The function `tpool_init()` in Fig. 6.6 initializes the task pool by allocating the data structure and initializing it with the argument values provided. Moreover, the threads used for the execution of the tasks are generated and their IDs are stored in `tpl->threads[i]` for `i=0,...,num_threads-1`. Each of these threads uses the function `tpool_thread()` as start function, see Fig. 6.7. This function has one argument specifying the task pool data structure to be used. Task execution is performed in an infinite loop. In each iteration of the loop, a task is retrieved from the head of the task list. If the task list is empty, the executing thread is blocked on the condition variable `not_empty` as described above. Otherwise, a task `wl` is retrieved from the list. If the task pool has been full before the retrieval, all threads blocked on `not_full`, waiting to insert a task, are woken up using

```
typedef struct work {
  void (*routine)();
  void *arg;
  struct work *next;
} work_t;

typedef struct tpool {
  int num_threads, max_size, current_size;
  pthread_t *threads;
  work_t *head, *tail;
  pthread_mutex_t lock;
  pthread_cond_t not_empty, not_full;
} tpool_t;

tpool_t *tpool_init (int num_threads, int max_size) {
  int i;
  tpool_t *tpl;

  tpl = (tpool_t *) malloc (sizeof (tpool_t));
  tpl->num_threads = num_threads;
  tpl->max_size = max_size;
  tpl->current_size = 0;
  tpl->head = tpl->tail = NULL;

  pthread_mutex_init (&(tpl->lock), NULL);
  pthread_cond_init (&(tpl->not_empty), NULL);
  pthread_cond_init (&(tpl->not_full), NULL);
  tpl->threads = (pthread_t *)malloc(sizeof(pthread_t)*num_threads);
  for (i=0; i<num_threads; i++)
    pthread_create (&(tpl->threads[i]), NULL,
                    tpool_thread, (void *) tpl);
  return tpl;
}
```

Fig. 6.6 Implementation of a task pool (part 1): The data structure `work_t` represents a task to be executed. The task pool data structure `tpool_t` contains a list of tasks with `head` pointing to the first element and `tail` pointing to the last element, as well as a set of threads `threads` to execute the tasks. The function `tpool_init()` is used to initialize a task pool data structure `tpl`

`pthread_cond_broadcast()`. The access to the task pool structure is protected by the mutex variable `tpl->lock`. The retrieved task `wl` is executed by calling the stored task function `wl->routine()` using the stored argument `wl->arg`. The execution of the retrieved task `wl` may lead to the generation of new tasks which are then inserted into the task pool using `tpool_insert()` by the executing thread.

The function `tpool_insert()` is used to insert tasks into the task pool. If the task pool is full when calling this function, the executing thread is blocked on the condition variable `not_full`. If the task pool is not full, a new task structure

```
void *tpool_thread (void *arg) {
  work_t *wl;
  tpool_t *tpl = (tpool_t *) arg;

  for( ; ; ) {
    pthread_mutex_lock (&(tpl->lock));
    while (tpl->current_size == 0)
      pthread_cond_wait (&(tpl->not_empty), &(tpl->lock));
    wl = tpl->head;
    tpl->current_size --;
    if (tpl->current_size == 0)
      tpl->head = tpl->tail = NULL;
    else tpl->head = wl->next;
    if (tpl->current_size == tpl->max_size - 1)
      pthread_cond_broadcast (&(tpl->not_full));
    pthread_mutex_unlock (&(tpl->lock));
    (*(wl->routine))(wl->arg);
    free(wl);
  }
}

void tpool_insert (tpool_t *tpl, void (*routine)(), void *arg) {
  work_t *wl;

  pthread_mutex_lock (&(tpl->lock));
  while (tpl->current_size == tpl->max_size)
    pthread_cond_wait (&(tpl->not_full), &(tpl->lock));
  wl = (work_t *) malloc (sizeof (work_t));
  wl->routine = routine;
  wl->arg = arg;
  wl->next = NULL;
  if (tpl->current_size == 0) {
          tpl->tail = tpl->head = wl;
          pthread_cond_signal (&(tpl->not_empty));
  }
  else {
          tpl->tail->next = wl;
          tpl->tail = wl;
  }
  tpl->current_size ++;
  pthread_mutex_unlock (&(tpl->lock));
}
```

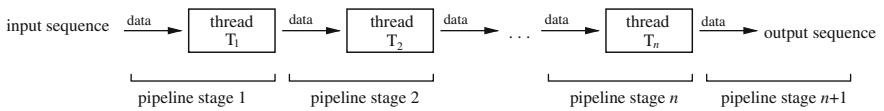
Fig. 6.7 Implementation of a task pool (part 2): The function `tpool_thread()` is used to extract and execute tasks. The function `tpool_insert()` is used to insert tasks into the task pool

is generated and filled and is inserted at the end of the task list. If the task pool has been empty before the insertion, one of the threads blocked on the condition variable `not_empty` is woken up using `pthread_cond_signal()`. The access to the task pool `tpl` is protected by the mutex variable `tpl->lock`.

The described implementation is especially suited for a master–slave model. A master thread uses `tpool_init()` to generate a number of slave threads each of which executes the function `tpool_thread()`. The tasks to be executed are defined according to the specific requirements of the application problem and are inserted in the task pool by the master thread using `tpool_insert()`. Tasks can also be inserted by the slave threads when the execution of a task leads to the generation of new tasks. After the execution of all tasks is completed, the master thread terminates the slave threads. To do so, the master thread wakes up all threads blocked on the condition variables `not_full` and `not_empty` and terminates them. Threads that are just executing a task are terminated as soon as they have finished the execution of this task.

6.1.7 Parallelism by Pipelining

In the pipelining model, a stream of data items is processed one after another by a sequence of threads $T_1, \ldots, T_n$ where each thread T_i performs a specific operation on each element of the data stream and passes the element onto the next thread T_{i+1}:



This results in an input/output relation between the threads: Thread T_i receives the output of thread T_{i+1} as input and produces data elements for thread T_{i+1}, $1 < i < n$. Thread T_1 reads the sequence of input elements, thread T_n produces the sequence of output elements. After a start-up phase with n steps, all threads can work in parallel and can be executed by different processors in parallel. The pipeline model requires some coordination between the cooperating threads: Thread T_i can start the computation of its corresponding stage only if the predecessor thread T_{i-1} has provided the input data element. Moreover, thread T_i can forward its output element to the successor thread T_{i+1}, only if T_{i+1} has finished its computation of the previous data item and is ready to receive a new data element.

The coordination of the threads of the pipeline stages can be organized with the help of condition variables. This will be demonstrated in the following for a simple example in which a sequence of integer values is incremented step by step in each pipeline stage, see also [25]. Thus, in each pipeline stage, the same computation is performed. But the coordination mechanism can also be applied if each pipeline stage performs a different computation.

```
typedef struct stage { // pipeline stage
  pthread_mutex_t m;
  pthread_cond_t avail; // input data available for this stage?
  pthread_cond_t ready; // stage ready to receive new data?
  int data_ready; // != 0, if other data is currently computed
  long data; // data element
  pthread_t thread; // Thread ID
  struct stage *next;
} stage_t;
typedef struct pipe { // pipeline
  pthread_mutex_t m;
  stage_t *head, *tail; // first/last stage of the pipeline
  int stages; // number of stages of the pipeline
  int active; // number of active data elements in the pipeline
} pipe_t;
const N_STAGES = 10;
```

Fig. 6.8 Implementation of a pipeline (part 1): data structures for the implementation of a pipeline model in Pthreads

For each stage of the pipeline, a data structure of type `stage_t` is used, see Fig. 6.8. This data structure contains a mutex variable `m` for synchronizing the access to the stage and two condition variables `avail` and `ready` for synchronizing the threads of neighboring stages. The condition variable `avail` is used to notify a thread that a data element is available to be processed by its pipeline stage. Thus, the thread can start the computation. A thread is blocked on the condition variable `avail` if no data element from the predecessor stage is available. The condition variable `ready` is used to notify the thread of the preceding pipeline stage that it can forward its output to the next pipeline stage. The thread of the preceding pipeline stage is blocked on this condition variable if it cannot directly forward its output data element to the next stage. The entry `data_ready` in the data structure for a stage is used to record whether a data element is currently available (value 1) for this pipeline stage or not (value 0). The entry `data` contains the actual data element to be processed. For the simple example discussed here, this is a single integer value, but this could be any data element for more complex applications. The entry `thread` is the TID of the thread used for this stage, and next is a reference to the next pipeline stage.

The entire pipeline is represented by the data structure `pipe_t` containing a mutex variable `m` and two pointers `head` and `tail` to the first and the last stages of the pipeline, respectively. The last stage of the pipeline is used to store the final result of the computation performed for each data element. There is no computation performed in this last stage, and there is no corresponding thread associated with this stage.

```
int pipe_send(stage_t *nstage, long data) {
  //parameter: target stage and data element to be processed
  pthread_mutex_lock(&nstage->m);
  {
     while (nstage->data_ready)
       pthread_cond_wait(&nstage->ready, &nstage->m);
     nstage->data = data;
     nstage->data_ready = 1;
     pthread_cond_signal(&nstage->avail);
  }
  pthread_mutex_unlock(&nstage->m);
}
void *pipe_stage(void *arg) {
  stage_t *stage = (stage_t*)arg;
  long result_data;
  pthread_mutex_lock(&stage->m);
  {
     for ( ; ; ) {
       while (!stage->data_ready) { // wait for data
         pthread_cond_wait(&stage->avail, &stage->m);
       }
       // process data element and forward to next stage :
       result_data = stage->data + 1; // compute result data element
       pipe_send(stage->next, result_data);
       stage->data_ready = 0; // processing finished
       pthread_cond_signal(&stage->ready);
     }
  }
  pthread_mutex_unlock(&stage->m); //this unlock will never be reached
}
```

Fig. 6.9 Implementation of a pipeline (part 2): functions to forward data elements to a pipeline stage and thread functions for the pipeline stages

The function `pipe_send()`, shown in Fig. 6.9, is used to send a data element to a stage of the pipeline. This function is used to send a data element to the first stage of the pipeline, and it is also used to pass a data element to the next stage of the pipeline after the computation of a stage has been completed. The stage receiving the data element is identified by the parameter `nstage`. Before inserting the data element, the mutex variable `m` of the receiving stage is locked to ensure that only one thread at a time is accessing the stage. A data element can be written into the receiving stage only if the computation of the previous data element in this stage has been finished. This is indicated by the condition `data_ready` = `0`. If this is not the case, the sending thread is blocked on the condition variable `ready` of the receiving stage. If the receiving stage is ready to receive the data element, the sending thread writes the element into the stage and wakes up the thread of the receiving stage if it is blocked on the condition variable `avail`.

Each of the threads participating in the pipeline computation executes the function `pipe_stage()`, see Fig. 6.9. The same function can be used for each stage for our example, since each stage performs the same computations. The function receives a pointer to its corresponding pipeline stage as an argument. A thread executing the function performs an infinite loop waiting for the arrival of data elements to be processed. The thread blocks on the condition variable `avail` if there is currently no data element available. If a data element is available, the thread performs its computation (increment by 1) and sends the result to the next pipeline stage `stage->next` using `pipe_send()`. Then it sends a notification to the thread associated with the next stage, which may be blocked on the condition variable `ready`. The notified thread can then continue its computation.

Thus, the synchronization of two neighboring threads is performed by using the condition variables `avail` and `ready` of the corresponding pipeline stages. The entry `data_ready` is used for the condition and determines which of the two threads is blocked and woken up. The entry of a stage is set to 0 if the stage is ready to receive a new data element to be processed by the associated thread. The entry `data_ready` of the next stage is set to 1 by the associated thread of the preceding stage if a new data element has been put into the next stage and is ready to be processed. In the simple example given here, the same computations are performed in each stage, i.e., all corresponding threads execute the same function `pipe_stage()`. For more complex scenarios, it is also possible that the different threads execute different functions, thus performing different computations in each pipeline stage.

The generation of a pipeline with a given number of stages can be achieved by calling the function `pipe_create()`, see Fig. 6.10. This function generates and initializes the data structures for the representation of the different stages. An additional stage is generated to hold the final result of the pipeline computation, i.e., the total number of stages is `stages+1`. For each stage except for the last additional stage, a thread is created. Each of these threads executes the function `pipe_stage()`.

The function `pipe_start()` is used to transfer a data element to the first stage of the pipeline, see Fig. 6.10. The actual transfer of the data element is done by calling the function `pipe_send()`. The thread executing `pipe_start()` does not wait for the result of the pipeline computation. Instead, `pipe_start()` returns control immediately. Thus, the pipeline works asynchronously with the thread which transfers data elements to the pipeline for computation. The synchronization between this thread and the thread of the first pipeline stage is performed within the function `pipe_send()`.

The function `pipe_result()` is used to take a result value out of the last stage of the pipeline, see Fig. 6.11. The entry `active` in the pipeline data structure `pipe_t` is used to count the number of data elements currently stored in the different pipeline stages. For `pipe->active = 0`, no data element is stored in the pipeline. In this case, `pipe_result()` immediately returns without providing a data element. For `pipe->active > 0`, `pipe_result()` is blocked on the condition variable `avail` of the last pipeline stage until a data element arrives at

```
int pipe_create(pipe_t *pipe, int stages) { // creation of the pipeline
  int pi;
  stage_t **link = &pipe->head;
  stage_t *new_stage, *stage;
  pthread_mutex_init(&pipe->m, NULL);
  pipe->stages = stages;
  pipe->active = 0;
  // create stages+1 stages, last stage is for the result
  for (pi = 0; pi <= stages; pi++) {
     new_stage = (stage_t *)malloc(sizeof(stage_t));
     pthread_mutex_init(&new_stage->m, NULL);
     pthread_cond_init(&new_stage->avail, NULL);
     pthread_cond_init(&new_stage->ready, NULL);
     new_stage->data_ready = 0;
     *link = new_stage; // make list link
     link = &new_stage->next;
  }
  *link = (stage_t *) NULL;
  pipe->tail = new_stage;
  // create a thread for each stage except the last one
  for (stage = pipe->head; stage->next != NULL; stage = stage->next){
     pthread_create(&stage->thread, NULL, pipe_stage, (void*)stage);
  }
}
int pipe_start(pipe_t *pipe, long v) { // start pipeline computation
  pthread_mutex_lock(&pipe->m);
  {
     pipe->active++;
  }
  pthread_mutex_unlock(&pipe->m);
  pipe_send(pipe->head, v);
}
```

Fig. 6.10 Implementation of a pipeline (part 3): Pthreads functions to generate and start a pipeline computation

this stage. This happens if the thread associated with the next to the last stage uses `pipe_send()` to transfer a processed data element to the last pipeline stage, see Fig. 6.9. By doing so, this thread wakes up a thread that is blocked on the condition variable `avail` of the last stage, if there is a thread waiting. If so, the woken-up thread is the one which tries to take a result value out of the last stage using `pipe_result()`.

The main program of the pipeline example is given in Fig. 6.11. It first uses `pipe_create()` to generate a pipeline with a given number of stages. Then it reads from `stdin` lines with numbers, which are the data elements to be processed. Each such data element is forwarded to the first stage of the pipeline using `pipe_start()`. Doing so, the executing main thread may be blocked on the condition variable `ready` of the first stage until the stage is ready to receive the data

```
int pipe_result(pipe_t *pipe, long *result) {
  stage_t *tail = pipe->tail;
  int empty = 0;
  pthread_mutex_lock(&pipe->m);
  {
     if (pipe->active <= 0) empty = 1; // empty pipeline
     else pipe->active--; // remove a data element
  }
  pthread_mutex_unlock(&pipe->m);
  if (empty) return 0;
  pthread_mutex_lock(&tail->m);
  {
     while (!tail->data_ready) // wait for data element
       pthread_cond_wait(&tail->avail, &tail->m);
     *result = tail->data;
     tail->data_ready = 0;
     pthread_cond_signal(&tail->ready);
  }
  pthread_mutex_unlock(&tail->m);
  return 1;
}
int main(int argc, char *argv[]) {
  pipe_t pipe;
  long value, result;
  char line[128];
  pipe_create(&pipe, N_STAGES);
  // terminate program for input error or EOF
  while (fgets(line, sizeof(line), stdin)) {
     if (*line == '\0')
       continue; // ignore empty input lines
     if (!strcmp(line, "=")) {
       if (pipe_result(&pipe, &result))
         printf("%ld\n", result);
       else printf("ERROR: Pipe empty\n");
     }
     else {
       if (sscanf(line, "%ld", &value) < 1)
         printf("ERROR: Not an int\n");
       else pipe_start(&pipe, value);
     }
  }
  return 0;
}
```

Fig. 6.11 Implementation of a pipeline (part 4): main program and Pthreads function to remove a result element from the pipeline

element. An input line with a single character '=' causes the main thread to call `pipe_result()` to take a result element out of the last stage, if present.

Figure 6.12 illustrates the synchronization between neighboring pipeline threads as well as between the main thread and the threads of the first or the next to last stage for a pipeline with three stages and two pipeline threads T_1 and T_2. The figure shows the relevant entries of the data structure `stage_t` for each stage. The order of the access and synchronization operations performed by the pipeline threads is determined by the statements in `pipe_stage()` and is illustrated by circled

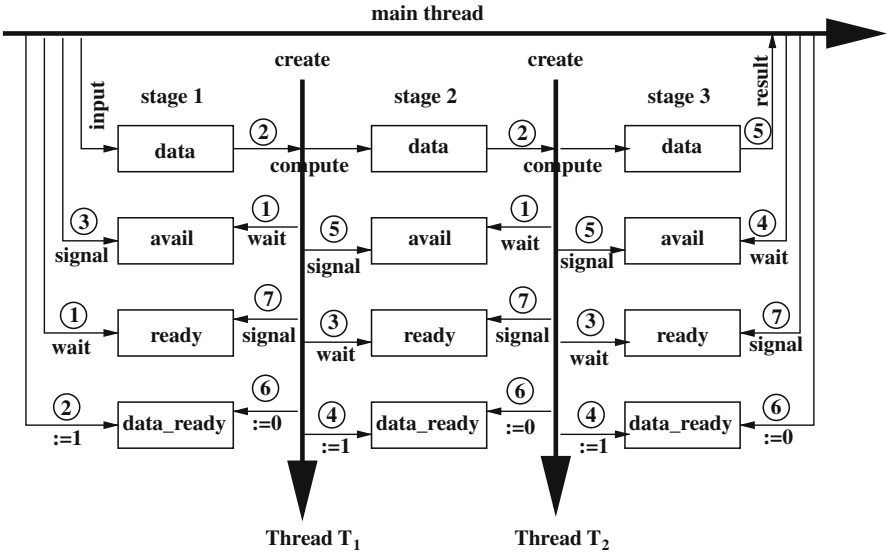


Fig. 6.12 Illustration of the synchronization between the pipeline threads for a pipeline with two pipeline threads and three stages, from the view of the data structures used. The *circled numbers* describe the order in which the synchronization steps are executed by the different threads according to the corresponding thread functions

numbers. The access and synchronization operations of the main thread result from the statements in `pipe_start()` and `pipe_result()`.

6.1.8 Implementation of a Client–Server Model

In a client–server system, we can distinguish between client threads and server threads. In a typical scenario, there are several server threads and several client threads. Server threads process requests that have been issued by the client threads. Client threads often represent the interface to the users of a system. During the processing of a request by a server thread, the issuing client thread can either wait for the request to be finished or can perform other operations, working concurrently with the server, and can collect the result at a later time when it is required. In the following, we illustrate the implementation of a client–server model for a simple example, see also [25].

Several threads repeatedly read input lines from `stdin` and output result lines to `stdout`. Before reading, a thread outputs a prompt to indicate which input is expected from the user. Server threads can be used to ensure the synchronization between the output of a prompt and the reading of the corresponding input line so that no output of another thread can occur in between. Client threads forward requests to the server threads to output a prompt or to read an input line. The server threads are terminated by a specific QUIT command. Figure 6.13 shows

```
#define REQ_READ 1
#define REQ_WRITE 2
#define REQ_QUIT 3
#define PROMPT_SIZE 32
#define TEXT_SIZE 128
typedef struct request {
  struct _request_t *next; // linked list
  int op;
  int synchronous; // 1 iff client waits for server
  int done_flag;
  pthread_cond_t done;
  char prompt[PROMPT_SIZE], text[TEXT_SIZE];
} request_t;
typedef struct tty_server { // data structure for server context
  request_t *first, *last;
  int running; // != 0, if server is running
  pthread_mutex_t m;
  pthread_cond_t request;
} tty_server_t;
#define TTY_SERVER_INITIALIZER { NULL, NULL, 0,
  PTHREAD_MUTEX_INITIALIZER,
  PTHREAD_COND_INITIALIZER }
tty_server_t tty_server = TTY_SERVER_INITIALIZER;
int client_threads;
pthread_mutex_t client_mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t client_done = PTHREAD_COND_INITIALIZER;
pthread_t server_thread;
```

Fig. 6.13 Implementation of a client–server system (part 1): data structure for the implementation of a client–server model with Pthreads

the data structures used for an implementation with Pthreads. The data structure `request_t` represents requests from the clients for the servers. The entry `op` specifies the requested operation to be performed (`REQ_READ`, `REQ_WRITE`, or `REQ_QUIT`). The entry `synchronous` indicates whether the client waits for the termination of the request (value 1) or not (value 0). The condition variable `done` is used for the synchronization between client and server, i.e., the client thread is blocked on `done` to wait until the server has finished the execution of the request. The entries `prompt` and `text` are used to store a prompt to be output or a text read in by the server, respectively. The data structure `tty_server_t` is used to store the requests sent to a server. The requests are stored in a FIFO (*first-in, first-out*) queue which can be accessed by `first` and `last`. The server thread is blocked on the condition variable `request` if the request queue is empty. The entry `running` indicates whether the corresponding server is running (value 1) or not (value 0).

```
void *tty_server_routine(void *arg) {
  static pthread_mutex_t prompt_mutex = PTHREAD_MUTEX_INITIALIZER;
  request_t *request;
  int op, len;
  for (;;) {
    pthread_mutex_lock(&tty_server.m);
    {
       while (tty_server.first == NULL)
         pthread_cond_wait(&tty_server.request, &tty_server.m);
       request = tty_server.first;
       tty_server.first = request->next;
       if (tty_server.first == NULL)
         tty_server.last = NULL;
    }
    pthread_mutex_unlock(&tty_server.m);
    switch (request->op) {
    case REQ_READ:
       puts(request->prompt);
       if (fgets(request->text, TEXT_SIZE, stdin) == NULL)
         request->text[0] = '\0';
       len = strlen(request->text);
       if (len > 0 && request->text[len - 1] == '\n')
         request->text[len - 1] = '\0';
       break;
    case REQ_WRITE:
       puts(request->text); break;
    default: // auch REQ_QUIT
       break;
    }
    op = request->op;
    if (request->synchronous) {
       pthread_mutex_lock(&tty_server.m);
       request->done_flag = 1;
       pthread_cond_signal(&request->done);
       pthread_mutex_unlock(&tty_server.m);
    }
    else free (request);
    if (op == REQ_QUIT) break;
  }
  return NULL;
}
```

Fig. 6.14 Implementation of a client–server system (part 2): server thread to process client requests

```
void tty_server_request(int op, int sync, char *prompt, char *string) {
  request_t *request;
  pthread_mutex_lock(&tty_server.m);
  {
    if (!tty_server.running) {
      pthread_create(&server_thread,NULL,tty_server_routine,NULL);
      tty_server.running = 1;
    }
    request = (request_t *) malloc(sizeof(request_t));
    request->op = op;
    request->synchronous = sync;
    request->next = NULL;
    if (sync) {
      request->done_flag = 0;
      pthread_cond_init(&request->done, NULL);
    }
    if (prompt != NULL) {
      strncpy(request->prompt, prompt, PROMPT_SIZE);
      request->prompt[PROMPT_SIZE - 1] = '\0';
    }
    else request->prompt[0] = '\0';
    if (op == REQ_WRITE && string != NULL) {
      strncpy(request->text, string, TEXT_SIZE);
      request->text[TEXT_SIZE - 1] = '\0';
    }
    else request->text[0] = '\0';
    if (tty_server.first == NULL)
      tty_server.first = tty_server.last = request;
    else {
      tty_server.last->next = request;
      tty_server.last = request;
    }
    pthread_cond_signal(&tty_server.request);
    if (sync) {
      while (!request->done_flag)
        pthread_cond_wait(&request->done, &tty_server.m);
      if (op == REQ_READ)
        strcpy(string, request->text);
      pthread_cond_destroy(&request->done);
      free(request);
    }
  }
  pthread_mutex_unlock(&tty_server.m);
}
```

Fig. 6.15 Implementation of a client–server system (part 3): forwarding of a request to the server thread

The program described in the following works with a single server thread, but can in principle be extended to an arbitrary number of servers.

The server thread executes the function `tty_server_routine()`, see Fig. 6.14. The server is blocked on the condition variable `request` as long as there are no requests to be processed. If there are requests, the server removes the first request from the queue and executes the operation (`REQ_READ`, `REQ_WRITE`, or `REQ_QUIT`) specified in the request. For the `REQ_READ` operation, the prompt specified with the request is output and a line is read in and stored into the `text` entry of the request structure. For a `REQ_WRITE` operation, the line stored in the `text` entry is written to `stdout`. The operation `REQ_QUIT` causes the server to finish its execution. If an issuing client waits for the termination of a request (entry `synchronous`), it is blocked on the condition variable `done` in the corresponding request structure. In this case, the server thread wakes up the blocked client thread using `pthread_cond_signal()` after the request has been processed. For asynchronous requests, the server thread is responsible to free the request data structure.

The client threads use the function `tty_server_request()` to forward a request to the server, see Fig. 6.15. If the server thread is not running yet, it will be started in `tty_server_request()`. The function allocates a request structure of type `request_t` and initializes it according to the requested operation. The request structure is then inserted into the request queue of the server. If the server is blocked waiting for requests to arrive, it is woken up using `pthread_cond_signal()`. If the client wants to wait for the termination of the request by the server, it is blocked on the condition variable `done` in the request structure, waiting for the server to wake it up again. The client threads execute the function `client_routine()`, see Fig. 6.16. Each client sends read and write requests to the server using the function `tty_server_request()` until the user terminates the client thread by specifying an empty line as input. When the last client thread has been terminated, the main thread which is blocked on the condition variable `client_done` is woken up again. The main thread generates the client threads and then waits until all client threads have been terminated. The server thread is not started by the main thread, but by the client thread which sends the first request to the server using `tty_server_routine()`. After all client threads are terminated, the server thread is terminated by the main thread by sending a `REQ_QUIT` request.

6.1.9 Thread Attributes and Cancellation

Threads are created using `pthread_create()`. In the previous sections, we have specified `NULL` as the second argument, thus leading to the generation of threads with default characteristics. These characteristics can be changed with the help of attribute objects. To do so, an attribute object has to be allocated and initialized before using the attribute object as parameter of `pthread_create()`. An attribute object for threads has type `pthread_attr_t`. Before an attribute object can be used, it must first be initialized by calling the function

```
void *client_routine(void *arg) {
  int my_nr = *(int*)arg, loops;
  char prompt[PROMPT_SIZE], string[TEXT_SIZE];
  char format[TEXT_SIZE + 64];
  sprintf(prompt, "Client %d>", my_nr);
  for (;;) {
    tty_server_request(REQ_READ, 1, prompt, string);
    // synchronized input
    if (string[0] == '\0') break; // program exit
    for (loops = 0; loops < 4; loops++) {
       sprintf(format, "(%d # %d)%s", my_nr, loops, string);
       tty_server_request(REQ_WRITE, 0, NULL, format);
       sleep(1);
    }
  }
  pthread_mutex_lock(&client_mutex);
  client_threads--;
  if (client_threads == 0) pthread_cond_signal(&client_done);
  pthread_mutex_unlock(&client_mutex);
  return NULL;
}
#define N_THREADS 4
int main(int argc, char *argv[]) {
  pthread_t thread;
  int i;
  int args[N_THREADS];
  client_threads = N_THREADS;
  pthread_mutex_lock(&client_mutex);
  {
    for (i = 0; i < N_THREADS; i++) {
       args[i] = i;
       pthread_create(&thread, NULL, client_routine, &args[i]);
    }
    while (client_threads > 0)
       pthread_cond_wait(&client_done, &client_mutex);
  }
  pthread_mutex_unlock(&client_mutex);
  printf("All clients done\n");
  tty_server_request(REQ_QUIT, 1, NULL, NULL);
  return 0;
}
```

Fig. 6.16 Implementation of a client–server system (part 4): client thread and main thread

```
int pthread_attr_init (pthread_attr_t *attr).
```

This leads to an initialization with the default attributes, corresponding to the default characteristics. By changing an attribute value, the characteristics can be changed. Pthreads provide attributes to influence the return value of threads, setting the size and address of the runtime stack, or the cancellation behavior of the thread. For each attribute, Pthreads define functions to get and set the current attribute value. But Pthreads implementations are not required to support the modification of all attributes. In the following, the most important aspects are described.

6.1.9.1 Return Value

An important property of a thread is its behavior concerning thread termination. This is captured by the attribute `detachstate`. This attribute can be influenced by all Pthreads libraries. By default, the runtime system assumes that the return value of a thread T_1 may be used by another thread after the termination of T_1. Therefore, the internal data structure maintained for a thread will be kept by the runtime system after the termination of a thread until another thread retrieves the return value using `pthread_join()`, see Sect. 6.1.1. Thus, a thread may bind resources even after its termination. This can be avoided if the programmer knows in advance that the return value of a thread will not be needed. If so, the thread can be generated such that its resources are immediately returned to the runtime system after its termination. This can be achieved by changing the `detachstate` attribute. The following two functions are provided to get or set this attribute value:

```
int pthread_attr_getdetachstate (const pthread_attr_t *attr,
                                 int *detachstate)
int pthread_attr_setdetachstate (pthread_attr_t *attr,
                                 int detachstate).
```

The attribute value `detachstate=PTHREAD_CREATE_JOINABLE` means that the return value of the thread is kept until it is joined by another thread. The attribute value `detachstate=PTHREAD_CREATE_DETACHED` means that the thread resources are freed immediately after thread termination.

6.1.9.2 Stack Characteristics

The different threads of a process have a shared program and data memory and a shared heap, but each thread has its own runtime stack. For most Pthreads libraries, the size and address of the local stack of a thread can be changed, but it is not required that a Pthreads library support this option. The local stack of a thread is used to store local variables of functions whose execution has not yet been terminated. The size required for the local stack is influenced by the size of the local variables and the nesting depth of function calls to be executed. This size may be large for recursive functions. If the default stack size is too small, it can be

increased by changing the corresponding attribute value. The Pthreads library that is used supports this if the macro

```
_POSIX_THREAD_ATTR_STACKSIZE
```

is defined in <unistd.h>. This can be checked by

```
#ifdef _POSIX_THREAD_ATTR_STACKSIZE     or
if (sysconf (_SC_THREAD_ATTR_STACKSIZE) == -1)
```

in the program. If it is supported, the current stack size stored in an attribute object can be retrieved or set by calling the functions

```
int pthread_attr_getstacksize (const pthread_attr_t *attr,
                               size_t *stacksize)
int pthread_attr_setstacksize (pthread_attr_t *attr,
                               size_t stacksize).
```

Here, size_t is a data type defined in <unistd.h> which is usually implemented as unsigned int. The parameter stacksize is the size of the stack in bytes. The value of stacksize should be at least PTHREAD_STACK_MIN which is predefined by Pthreads as the minimum stack size required by a thread. Moreover, if the macro

```
_POSIX_THREAD_ATTR_STACKADDR
```

is defined in <unistd.h>, the address of the local stack of a thread can also be influenced. The following two functions

```
int pthread_attr_getstackaddr (const pthread_attr_t *attr,
                               size_t **stackaddr)
int pthread_attr_setstackaddr (pthread_attr_t *attr,
                               size_t *stackaddr)
```

are provided to get or set the current stack address stored in an attribute object. The modification of stack-related attributes should be used with caution, since such modification can result in non-portable programs. Moreover, the option is not supported by all Pthreads libraries.

After the modification of specific attribute values in an attribute object a thread with the chosen characteristics can be generated by specifying the attribute object as second parameter of pthread_create(). The characteristics of the new thread are defined by the attribute values stored in the attribute object at the time at which pthread_create() is called. These characteristics cannot be changed at a later time by changing attribute values in the attribute object.

6.1.9.3 Thread Cancellation

In some situations, it is useful to stop the execution of a thread from outside, e.g., if the result of the operation performed is no longer needed. An example could be an application where several threads are used to search in a data structure for a specific entry. As soon as the entry is found by one of the threads, all other threads can stop execution to save execution time. This can be reached by sending a cancellation request to these threads.

In Pthreads, a thread can send a cancellation request to another thread by calling the function

```
int pthread_cancel (pthread_t thread)
```

where `thread` is the thread ID of the thread to be terminated. A call of this function does not necessarily lead to an immediate termination of the specified target thread. The exact behavior depends on the cancellation type of this thread. In any case, control immediately returns to the calling thread, i.e., the thread issuing the cancellation request does not wait for the cancelled thread to be terminated. By default, the cancellation type of the thread is **deferred**. This means that the thread can only be cancelled at specific **cancellation points** in the program. After the arrival of a cancellation request, thread execution continues until the next cancellation point is reached. The Pthreads standard defines obligatory and optional cancellation points. Obligatory cancellation points typically include all functions at which the executing thread may be blocked for a substantial amount of time. Examples are `pthread_cond_wait()`, `pthread_cond_timedwait()`, `open()`, `read()`, `wait()`, or `pthread_join()`, see [25] for a complete list. Optional cancellation points include many file and I/O operations. The programmer can insert additional cancellation points into the program by calling the function

```
void pthread_testcancel().
```

When calling this function, the executing thread checks whether a cancellation request has been sent to it. If so, the thread is terminated. If not, the function has no effect. Similarly, at predefined cancellation points the executing thread also checks for cancellation requests. A thread can set its cancellation type by calling the function

```
int pthread_setcancelstate (int state, int *oldstate).
```

A call with `state = PTHREAD_CANCEL_DISABLE` disables the cancelability of the calling thread. The previous cancellation type is stored in `*oldstate`. If the cancelability of a thread is disabled, it does not check for cancellation requests when reaching a cancellation point or when calling `pthread_testcancel()`, i.e., the thread cannot be cancelled from outside. The cancelability of a thread can

be enabled again by calling `pthread_setcancelstate()` with the parameter value `state = PTHREAD_CANCEL_ENABLE`.

By default, the cancellation type of a thread is deferred. This can be changed to **asynchronous cancellation** by calling the function

```
int pthread_setcanceltype (int type, int *oldtype)
```

with `type=PTHREAD_CANCEL_ASYNCHRONOUS`. This means that this thread can be cancelled not only at cancellation points. Instead, the thread is terminated immediately after the cancellation request arrives, even if the thread is just performing computations within a critical section. This may lead to inconsistent states causing errors for other threads. Therefore, asynchronous cancellation may be harmful and should be avoided. Calling `pthread_setcanceltype()` with `type = PTHREAD_CANCEL_DEFERRED` sets a thread to the usual deferred cancellation type.

6.1.9.4 Cleanup Stack

In some situations, a thread may need to restore some state when it is cancelled. For example, a thread may have to release a mutex variable when it is the owner before being cancelled. To support such state restorations, a cleanup stack is associated with each thread, containing function calls to be executed just before thread cancellation. These function calls can be used to establish a consistent state at thread cancellation, e.g., by unlocking mutex variables that have previously been locked. This is necessary if there is a cancellation point between acquiring and releasing a mutex variable. If a cancellation happens at such a cancellation point without releasing the mutex variable, another thread might wait forever to become the owner. To avoid such situations, the cleanup stack can be used: When acquiring the mutex variable, a function call (cleanup handler) to release it is put onto the cleanup stack. This function call is executed when the thread is cancelled. A cleanup handler is put onto the cleanup stack by calling the function

```
void pthread_cleanup_push (void (*routine) (void *), void *arg)
```

where `routine` is a pointer to the function used as cleanup handler and `arg` specifies the corresponding argument values. The cleanup handlers on the cleanup stack are organized in LIFO (*last-in, first-out*) order, i.e., the handlers are executed in the opposite order of their placement, beginning with the most recently added handler. The handlers on the cleanup stack are automatically executed when the corresponding thread is cancelled or when it exits by calling `pthread_exit()`. A cleanup handler can be removed from the cleanup stack by calling the function

```
void pthread_cleanup_pop (int execute).
```

This call removes the most recently added handler from the cleanup stack. For `execute`$\neq$`0`, this handler will be executed when it is removed. For `execute=0`, this handler will be removed without execution. To produce portable programs, corresponding calls of `pthread_cleanup_push()` and `pthread_cleanup_pop()` should be organized in pairs within the same function.

Example To illustrate the use of cleanup handlers, we consider the implementation of a semaphore mechanism in the following. A *(counting) semaphore* is a data type with a counter which can have non-negative integer values and which can be modified by two operations: A *signal* operation increments the counter and wakes up a thread which is blocked on the semaphore, if there is such a thread; a *wait* operation blocks the executing thread until the counter has a value > 0, and then decrements the counter. Counting semaphores can be used for the management of limited resources. In this case, the counter is initialized to the number of available resources. *Binary semaphores*, on the other hand, can only have value 0 or 1. They can be used to ensure mutual exclusion when executing critical sections.

Figure 6.17 illustrates the use of cleanup handlers to implement a semaphore mechanism based on condition variables, see also [143]. A semaphore is represented by the data type `sema_t`. The function `AcquireSemaphore()` waits until the counter has values > 0, before decrementing the counter. The function `Release`

```
typedef struct Sema {
  pthread_mutex_t mutex;
  pthread_cond_t cond;
  int count;
} sema_t;

void CleanupHandler (void *arg)
{ pthread_mutex_unlock ((pthread_mutex_t *) arg);}

void AquireSemaphore (sema_t *ps)
{
  pthread_mutex_lock (&(ps->mutex));
  pthread_cleanup_push (CleanupHandler, &(ps->mutex));
  while (ps->count == 0)
    pthread_cond_wait (&(ps->cond), &(ps->mutex));
  --ps->count;
  pthread_cleanup_pop (1);
}

void ReleaseSemaphore (sema_t *ps)
{
  pthread_mutex_lock (&(ps->mutex));
  pthread_cleanup_push (CleanupHandler, &(ps->mutex));
  ++ps->count;
  pthread_cond_signal (&(ps->cond));
  pthread_cleanup_pop (1);
}
```

Fig. 6.17 Use of a cleanup handler for the implementation of a semaphore mechanism. The function `AquireSemaphore()` implements the access to the semaphore. The call of `pthread_cond_wait()` ensures that the access is performed not before the value `count` of the semaphore is larger than zero. The function `ReleaseSemaphore()` implements the release of the semaphore

`Semaphore()` increments the counter and then wakes up a waiting thread using `pthread_cond_signal()`. The access to the semaphore data structure is protected by a mutex variable in both cases, to avoid inconsistent states by concurrent accesses. At the beginning, both functions call `pthread_mutex_lock()` to lock the mutex variable. At the end, the call `pthread_cleanup_pop(1)` leads to the execution of `pthread_mutex_unlock()`, thus releasing the mutex variable again. If a thread is blocked in `AcquireSemaphore()` when executing the function `pthread_cond_wait(&(ps->cond),&(ps->mutex))` it implicitly releases the mutex variable `ps->mutex`. When the thread is woken up again, it first tries to become owner of this mutex variable again. Since `pthread_cond_wait()` is a cancellation point, a thread might be cancelled while waiting for the condition variable `ps->cond`. In this case, the thread first becomes the owner of the mutex variable before termination. Therefore, a cleanup handler is used to release the mutex variable again. This is obtained by the function `Cleanup_Handler()` in Fig. 6.17. □

6.1.9.5 Producer–Consumer Threads

The semaphore mechanism from Fig. 6.17 can be used for the synchronization between producer and consumer threads, see Fig. 6.18. A producer thread inserts entries into a buffer of fixed length. A consumer thread removes entries from the buffer for further processing. A producer can insert entries only if the buffer is not full. A consumer can remove entries only if the buffer is not empty. To control this, two semaphores `full` and `empty` are used. The semaphore `full` counts the number of occupied entries in the buffer. It is initialized to 0 at program start. The semaphore `empty` counts the number of free entries in the buffer. It is initialized to the buffer capacity. In the example, the buffer is implemented as an array of length 100, storing entries of type `ENTRY`. The corresponding data structure `buffer` also contains the two semaphores `full` and `empty`.

As long as the buffer is not full, a producer thread produces entries and inserts them into the shared buffer using `produce_item()`. For each insert operation, `empty` is decremented by using `AcquireSemaphore()` and `full` is incremented by using `ReleaseSemaphore()`. If the buffer is full, a producer thread will be blocked when calling `AcquireSemaphore()` for `empty`. As long as the buffer is not empty, a consumer thread removes entries from the buffer and processes them using `comsume_item()`. For each remove operation, `full` is decremented using `AcquireSemaphore()` and `empty` is incremented using `ReleaseSemaphore()`. If the buffer is empty, a consumer thread will be blocked when calling the function `AcquireSemaphore()` for `full`. The internal buffer management is hidden in the functions `produce_item()` and `consume_item()`.

After a producer thread has inserted an entry into the buffer, it wakes up a consumer thread which is waiting for the semaphore `full` by calling the function `ReleaseSemaphore(&buffer.full)`, if there is such a waiting consumer.

```
struct linebuf {
  ENTRY line[100];
  sema_t full, empty;
} buffer;

void *Producer(void *arg)
{
  while (1) {
    AquireSemaphore (&buffer.empty);
    produce_item();
    ReleaseSemaphore (&buffer.full);
  }
}

void *Consumer(void *arg)
{
  while (1) {
    AquireSemaphore (&buffer.full);
    consume_item();
    ReleaseSemaphore (&buffer.empty);
  }
}

void CreateSemaphore (sema_t *ps, int count)
{
  ps->count = count;
  pthread_mutex_init (&ps->mutex, NULL);
  pthread_cond_init (&ps->cond, NULL);
}

int main()
{
  pthread_t threadID[2];
  int i;
  void *status;

  CreateSemaphore (&buffer.empty, 100);
  CreateSemaphore (&buffer.full, 0);

  pthread_create (&threadID[0], NULL, Consumer, NULL);
  pthread_create (&threadID[1], NULL, Producer, NULL);

  for (i=0; i<2; i++)
    pthread_join (threadID[i], &status);
}
```

Fig. 6.18 Implementation of producer–consumer threads using the semaphore operations from Fig. 6.17

After a consumer has removed an entry from the buffer, it wakes up a producer which is waiting for `empty` by calling `ReleaseSemaphore(&buffer.empty)`, if there is such a waiting producer. The program in Fig. 6.18 uses one producer and one consumer thread, but it can easily be generalized to an arbitrary number of producer and consumer threads.

6.1.10 Thread Scheduling with Pthreads

The user threads defined by the programmer for each process are mapped to kernel threads by the library scheduler. The kernel threads are then brought to execution on the available processors by the scheduler of the operating system. For many Pthreads libraries, the programmer can influence the mapping of user threads to kernel threads using **scheduling attributes**. The Pthreads standard specifies a scheduling interface for this, but this is not necessarily supported by all Pthreads libraries. A specific Pthreads library supports the scheduling programming interface, if the macro `POSIX_THREAD_PRIORITY_SCHEDULING` is defined in `<unistd.h>`. This can also be checked dynamically in the program using `sysconf()` with parameter `_SC_THREAD_PRIORITY_SCHEDULING`. If the scheduling programming interface is supported and shall be used, the header file `<sched.h>` must be included into the program.

Scheduling attributes are stored in data structures of type `struct sched_param` which must be provided by the Pthreads library if the scheduling interface is supported. This type must at least have the entry

```
int sched_priority;
```

The scheduling attributes can be used to assign scheduling priorities to threads and to define scheduling policies and scheduling scopes. This can be set when a thread is created, but it can also be changed dynamically during thread execution.

6.1.10.1 Explicit Setting of Scheduling Attributes

In the following, we first describe how scheduling attributes can be set explicitly at thread creation.

The **scheduling priority** of a thread determines how privileged the library scheduler treats the execution of a thread compared to other threads. The priority of a thread is defined by an integer value which is stored in the `sched_priority` entry of the `sched_param` data structure and which must lie between a minimum and maximum value. These minimum and maximum values allowed for a specific scheduling policy can be determined by calling the functions

```
int sched_get_priority_min (int policy)
int sched_get_priority_max (int policy)
```

where `policy` specifies the scheduling policy. The minimum or maximum priority values are given as return value of these functions. The library scheduler maintains for each priority value a separate queue of threads with this priority that are ready for execution. When looking for a new thread to be executed, the library scheduler accesses the thread queue with the highest priority that is not empty. If this queue contains several threads, one of them is selected for execution according to

the scheduling policy. If there are always enough executable threads available at each point in program execution, it can happen that threads of low priority are not executed for quite a long time. The two functions

```
int pthread_attr_getschedparam (const pthread_attr_t *attr,
                                struct sched_param *param)
int pthread_attr_setschedparam (pthread_attr_t *attr,
                                const struct sched_param *param)
```

can be used to extract or set the priority value of an attribute data structure `attr`. To set the priority value, the entry `param->sched_priority` must be set to the chosen priority value before calling `pthread_attr_setschedparam()`.

The scheduling policy of a thread determines how threads of the same priority are executed and share the available resources. In particular, the scheduling policy determines how long a thread is executed if it is selected by the library scheduler for execution. Pthreads support three different scheduling policies:

- **SCHED_FIFO** (*first-in, first-out*): The executable threads of the same priority are stored in a FIFO queue. A new thread to be executed is selected from the beginning of the thread queue with the highest priority. The selected thread is executed until it either exits or blocks or until a thread with a higher priority becomes ready for execution. In the latter case, the currently executed thread with lower priority is interrupted and stored at the *beginning* of the corresponding thread queue. Then, the thread of higher priority starts execution. If a thread that has been blocked, e.g., waiting on a condition variable, becomes ready for execution again, it is stored at the *end* of the thread queue of its priority. If the priority of a thread is dynamically changed, it is stored at the *end* of the thread queue with the new priority.
- **SCHED_RR** (*round robin*): The thread management is similar to the policy SCHED_FIFO. The difference is that each thread is allowed to run for only a fixed amount of time, given by a predefined timeslice interval. After the interval has elapsed, and another thread of the same priority is ready for execution, the running thread will be interrupted and put at the `end` of the corresponding thread queue. The timeslice intervals are defined by the library scheduler. All threads of the same process use the same timeslice interval. The length of a timeslice interval of a process can be queried with the function

  ```
  int sched_rr_get_interval (pid_t pid, struct timespec *quantum)
  ```

  where `pid` is the process ID of the process. For `pid=0`, the information for that process is returned to the calling thread to which it belongs. The data structure of type `timespec` is defined as

  ```
  struct timespec { time_t tv_sec; long tv_nsec; } .
  ```

- **SCHED_OTHER**: Pthreads allow an additional scheduling policy, the behavior of which is not specified by the standard, but completely depends on the specific Pthreads library used. This allows the adaptation of the scheduling to a specific operating system. Often, a scheduling strategy is used which adapts the priorities of the threads to their I/O behavior, such that interactive threads get a higher priority as compute-intensive threads. This scheduling policy is often used as default for newly created threads.

The scheduling policy used for a thread is set when the thread is created. If the programmer wants to use a scheduling policy other than the default he can achieve this by creating an attribute data structure with the appropriate values and providing this data structure as argument for `pthread_create()`. The two functions

```
int pthread_attr_getschedpolicy (const pthread_attr_t *attr,
                                 int *schedpolicy)
int pthread_attr_setschedpolicy (pthread_attr_t *attr,
                                 int schedpolicy)
```

can be used to extract or set the scheduling policy of an attribute data structure `attr`. On some Unix systems, setting the scheduling policy may require superuser rights.

The **contention scope** of a thread determines which other threads are taken into consideration for the scheduling of a thread. Two options are provided: The thread may compete for processor resources with the threads of the corresponding process (*process contention scope*) or with the threads of all processes on the system (*system contention scope*). Two functions can be used to extract or set the contention scope of an attribute data structure `attr`:

```
int pthread_attr_getscope (const pthread_attr_t *attr,
                           int *contentionscope)
int pthread_attr_setscope (pthread_attr_t *attr,
                           int contentionscope).
```

The parameter value `contentionscope=PTHREAD_SCOPE_PROCESS` corresponds to a process contention scope, whereas a system contention scope can be obtained by the parameter value `contentionscope=PTHREAD_SCOPE_SYSTEM`. Typically, using a process contention scope leads to better performance than a system contention scope, since the library scheduler can switch between the threads of a process without calling the operating system, whereas switching between threads of different processes usually requires a call of the operating system, and this is usually relatively expensive [25]. A Pthreads library only needs to support one of the two contention scopes. If a call of `pthread_attr_setscope()` tries to set

a contention scope that is not supported by the specific Pthreads library, the error value ENOTSUP is returned.

6.1.10.2 Implicit Setting of Scheduling Attributes

Some application codes create a lot of threads for specific tasks. To avoid setting the scheduling attributes before each thread creation, Pthreads support the inheritance of scheduling information from the creating thread. The two functions

```
int pthread_attr_getinheritsched (const pthread_attr_t *attr,
                                  int *inheritsched)
int pthread_attr_setinheritsched (pthread_attr_t *attr,
                                  int inheritsched)
```

can be used to extract or set the inheritance status of an attribute data structure `attr`. Here, `inheritsched=PTHREAD_INHERIT_SCHED` means that a thread creation with this attribute structure generates a thread with the scheduling attributes of the creating thread, ignoring the scheduling attributes in the attribute structure. The parameter value `inheritsched=PTHREAD_EXPLICIT_SCHED` disables the inheritance, i.e., the scheduling attributes of the created thread must be set explicitly if they should be different from the default setting. The Pthreads standard does not specify a default value for the inheritance status. Therefore, if a specific behavior is required, the inheritance status must be set explicitly.

6.1.10.3 Dynamic Setting of Scheduling Attributes

The priority of a thread and the scheduling policy used can also be changed dynamically during the execution of a thread. The two functions

```
int pthread_getschedparam (pthread_t thread, int *policy,
                           struct sched_param *param)
int pthread_setschedparam (pthread_t thread, int policy,
                           const struct sched_param *param)
```

can be used to dynamically extract or set the scheduling attributes of a thread with TID `thread`. The parameter `policy` defines the scheduling policy; `param` contains the priority value.

Figure 6.19 illustrates how the scheduling attributes can be set explicitly before the creation of a thread. In the example, `SCHED_RR` is used as scheduling policy. Moreover, a medium priority value is used for the thread with ID `thread_id`. The inheritance status is set to `PTHREAD_EXPLICIT_SCHED` to transfer the scheduling attributes from `attr` to the newly created thread `thread_id`.

```
#include <unistd.h>
#include <pthread.h>
#include <sched.h>

void *thread_routine (void *arg) {return NULL;}

int main()
{
  pthread_t thread_id;
  pthread_attr_t attr;
  struct sched_param param;
  int policy, min_prio, max_prio;

  pthread_attr_init (&attr);
  if (sysconf (_SC_THREAD_PRIORITY_SCHEDULING) != -1) {
    pthread_attr_getschedpolicy (&attr, &policy);
    pthread_attr_getschedparam (&attr, &param);
    printf ("Default: Policy %d, Priority %d \n", policy,
             param.sched_priority);
    pthread_attr_setschedpolicy (&attr, SCHED_RR);
    min_prio = sched_get_priority_min (SCHED_RR);
    max_prio = sched_get_priority_max (SCHED_RR);
    param.sched_priority = (min_prio + max_prio)/2;
    pthread_attr_setschedparam (&attr, &param);
    pthread_attr_setinheritsched (&attr, PTHREAD_EXPLICIT_SCHED);
  }
  pthread_create (&thread_id, &attr, thread_routine, NULL);
  pthread_join (thread_id, NULL);
  return 0;
}
```

Fig. 6.19 Use of scheduling attributes to define the scheduling behavior of a generated thread

6.1.11 Priority Inversion

When scheduling several threads with different priorities, it can happen with an unsuitable order of synchronization operations that a thread of lower priority prevents a thread of higher priority from being executed. This phenomenon is called **priority inversion**, indicating that a thread of lower priority is running although a thread of higher priority is ready for execution. This phenomenon is illustrated in the following example, see also [126].

Example We consider the execution of three threads A, B, C with high, medium, and low priority, respectively, on a single processor competing for a mutex variable m. The threads perform at program points $t_1, \ldots, t_6$ the following actions, see

Point in time	Event	Thread A high priority	Thread B medium priority	Thread C low priority	Mutex variable m
t_1	Start	/	/	/	Free
t_2	Start C	/	/	Running	Free
t_3	C locks m	/	/	Running	Locked by C
t_4	Start A	Running	/	Ready for execution	Locked by C
t_5	A locks m	Blocked	/	Running	Locked by C
t_6	Start B	Blocked	Running	Ready for execution	Locked by C

Fig. 6.20 Illustration of a priority inversion

Fig. 6.20 for an illustration. After the start of the program at time t_1, thread C of low priority is started at time t_2. At time t_3, thread C calls `pthread_mutex_lock(m)` to lock m. Since m has not been locked before, C becomes the owner of m and continues execution. At time t_4, thread A of high priority is started. Since A has a higher priority than C, C is blocked and A is executed. The mutex variable m is still locked by C. At time t_5, thread A tries to lock m using `pthread_mutex_lock(m)`. Since m has already been locked by C, A blocks on m. The execution of C resumes. At time t_6, thread B of medium priority is started. Since B has a higher priority than C, C is blocked and B is executed. C is still the owner of m. If B does not try to lock m, it may be executed for quite some time, even if there is a thread A of higher priority. But A cannot be executed, since it waits for the release of m by C. But C cannot release m, since C is not executed. Thus, the processor is continuously executing B and not A, although A has a higher priority than B. □

Pthreads provide two mechanisms to avoid priority inversion: priority ceiling and priority inheritance. Both mechanisms are optional, i.e., they are not necessarily supported by each Pthreads library. We describe both mechanisms in the following.

6.1.11.1 Priority Ceiling

The mechanism of priority ceiling is available for a specific Pthreads library if the macro

```
_POSIX_THREAD_PRIO_PROTECT
```

is defined in `<unistd.h>`. If priority ceiling is used, each mutex variable gets a priority value. The priority of a thread is automatically raised to this *priority ceiling value* of a mutex variable, whenever the thread locks the mutex variable. The thread keeps this priority as long as it is the owner of the mutex variable. Thus, a thread X cannot be interrupted by another thread Y with a lower priority than the priority of the mutex variable as long as X is the owner of the mutex variable. The owning thread can therefore work without interruption and can release the mutex variable as soon as possible.

In the example given above, priority inversion is avoided with priority ceiling if a priority ceiling value is used which is equal to or larger than the priority of thread

A. In the general case, priority inversion is avoided if the highest priority at which a thread will ever be running is used as priority ceiling value.

To use priority ceiling for a mutex variable, it must be initialized appropriately using a mutex attribute data structure of type `pthread_mutex_attr_t`. This data structure must first be declared and initialized using the function

```
int pthread_mutex_attr_init(pthread_mutex_attr_t attr)
```

where `attr` is the mutex attribute data structure. The default priority protocol used for `attr` can be extracted by calling the function

```
int pthread_mutexattr_getprotocol(const pthread_mutex_attr_t
                *attr, int *prio)
```

which returns the protocol in the parameter `prio`. The following three values are possible for `prio`:

- `PTHREAD_PRIO_PROTECT`: the priority ceiling protocol is used;
- `PTHREAD_PRIO_INHERIT`: the priority inheritance protocol is used;
- `PTHREAD_PRIO_NONE`: none of the two protocols is used, i.e., the priority of a thread does not change if it locks a mutex variable.

The function

```
int pthread_mutexattr_setprotocol(pthread_mutex_attr_t *attr,
                    int prio)
```

can be used to set the priority protocol of a mutex attribute data structure `attr` where `prio` has one of the three values just described. When using the priority ceiling protocol, the two functions

```
int pthread_mutexattr_getprioceiling(const pthread_ mutex_attr_t
            *attr, int *prio)
int pthread_mutexattr_setprioceiling(pthread_mutex_ attr_t *attr,
                int prio)
```

can be used to extract or set the priority ceiling value stored in the attribute structure `attr`. The ceiling value specified in `prio` must be a valid priority value. After a mutex attributed data structure `attr` has been initialized and possibly modified, it can be used for the initialization of a mutex variable with the specified properties, using the function

```
pthread_mutex_init (pthread_mutex_t *m, pthread_mutexattr_t
                    *attr)
```

see also Sect. 6.1.2.

6.1.11.2 Priority Inheritance

When using the priority inheritance protocol, the priority of a thread which is the owner of a mutex variable is automatically raised, if a thread with a higher priority tries to lock the mutex variable and is therefore blocked on the mutex variable. In this situation, the priority of the owner thread is raised to the priority of the blocked thread. Thus, the owner of a mutex variable always has the maximum priority of all threads waiting for the mutex variable. Therefore, the owner thread cannot be interrupted by one of the waiting threads, and priority inversion cannot occur. When the owner thread releases the mutex variable again, its priority is decreased again to the original priority value.

The priority inheritance protocol can be used if the macro

```
_POSIX_THREAD_PRIO_INHERIT
```

is defined in `<unistd.h>`. If supported, priority inheritance can be activated by calling the function `pthread_mutexattr_setprotocol()` with parameter value `prio = PTHREAD_PRIO_INHERIT` as described above. Compared to priority ceiling, priority inheritance has the advantage that no fixed priority ceiling value has to be specified in the program. Priority inversion is avoided also for threads with unknown priority values. But the implementation of priority inheritance in the Pthreads library is more complicated and expensive and therefore usually leads to a larger overhead than priority ceiling.

6.1.12 Thread-Specific Data

The threads of a process share a common address space. Thus, global and dynamically allocated variables can be accessed by each thread of a process. For each thread, a private stack is maintained for the organization of function calls performed by the thread. The local variables of a function are stored in the private stack of the calling thread. Thus, they can only be accessed by this thread, if this thread does not expose the address of a local variable to another thread. But the lifetime of local variables is only the lifetime of the corresponding function activation. Thus, local variables do not provide a persistent thread-local storage. To use the value of a local variable throughout the lifetime of a thread, it has to be declared in the start function of the thread and passed as parameter to all functions called by this thread. But depending on the application, this would be quite tedious and would artificially increase the number of parameters. Pthreads supports the use of thread-specific data with an additional mechanism.

To generate thread-specific data, Pthreads provide the concept of *keys* that are maintained in a process-global way. After the creation of a key it can be accessed by each thread of the corresponding process. Each thread can associate thread-specific data to a key. If two threads associate different data to the same key, each of the two

threads gets only its own data when accessing the key. The Pthreads library handles the management and storage of the keys and their associated data.

In Pthreads, keys are represented by the predefined data type `pthread_key_t`. A key is generated by calling the function

```
int pthread_key_create (pthread_key_t *key,
                        void (*destructor)(void *)).
```

The generated key is returned in the parameter `key`. If the key is used by several threads, the address of a global variable or a dynamically allocated variable must be passed as `key`. The function `pthread_key_create()` should only be called once for each `pthread_key_t` variable. This can be ensured with the `pthread_once()` mechanism, see Sect. 6.1.4. The optional parameter `destructor` can be used to assign a deallocation function to the key to clean up the data stored when the thread terminates. If no deallocation is required, `NULL` should be specified. A key can be deleted by calling the function

```
int pthread_key_delete (pthread_key_t key).
```

After the creation of a key, its associated data is initialized to `NULL`. Each thread can associate new data `value` to the key by calling the function

```
int pthread_setspecific (pthread_key_t key, void *value).
```

Typically, the address of a *dynamically* generated data object will be passed as `value`. Passing the address of a local variable should be avoided, since this address is no longer valid after the corresponding function has been terminated. The data associated with a key can be retrieved by calling the function

```
void *pthread_getspecific (pthread_key_t key).
```

The calling thread always obtains the data value that it has previously associated with the key using `pthread_setspecific()`. When no data has been associated yet, `NULL` is returned. `NULL` is also returned, if another thread has associated data with the key, but not the calling thread. When a thread uses the function `pthread_setspecific()` to associate new data to a key, data that has previously been associated with this key by this thread will be overwritten and is lost.

An alternative to thread-specific data is the use of thread-local storage (TLS) which is provided since the C99 standard. This mechanism allows the declaration of variables with the storage class keyword `__thread` with the effect that each thread gets a separate instance of the variable. The instance is deleted as soon as the thread

terminates. The `__thread` storage class keyword can be applied to global variables and static variables. It cannot be applied to block-scoped automatic or non-static variables.

6.2 Java Threads

Java supports the development of multi-threaded programs at the language level. Java provides language constructs for the synchronized execution of program parts and supports the creation and management of threads by predefined classes. In this chapter, we demonstrate the use of Java threads for the development of parallel programs for a shared address space. We assume that the reader knows the principles of object-oriented programming as well as the standard language elements of Java. We concentrate on the mechanisms for the development of multi-threaded programs and describe the most important elements. We refer to [129, 113] for a more detailed description. For a detailed description of Java, we refer to [51].

6.2.1 Thread Generation in Java

Each Java program in execution consists of at least one thread of execution, the *main thread*. This is the thread which executes the `main()` method of the class which has been given to the Java Virtual Machine (JVM) as start argument.

More user threads can be created explicitly by the main thread or other user threads that have been started earlier. The creation of threads is supported by the predefined class `Thread` from the standard package `java.lang`. This class is used for the representation of threads and provides methods for the creation and management of threads.

The interface `Runnable` from `java.lang` is used to represent the program code executed by a thread; this code is provided by a `run()` method and is executed asynchronously by a separate thread. There are two possibilities to arrange this: inheriting from the `Thread` class or using the interface `Runnable`.

6.2.1.1 Inheriting from the `Thread` Class

One possibility to obtain a new thread is to define a new class `NewClass` which inherits from the predefined class `Thread` and which defines a method `run()` containing the statements to be executed by the new thread. The `run()` method defined in `NewClass` overwrites the predefined `run()` method from `Thread`.

The `Thread` class also contains a method `start()` which creates a new thread executing the given `run()` method.

The newly created thread is executed asynchronously with the generating thread. After the execution of `start()` and the creation of the new thread, the control will be immediately returned to the generating thread. Thus, the generating thread

Fig. 6.21 Thread creation by overwriting the `run()` method of the `Thread` class

```
import java.lang.Thread;
public class NewClass extends Thread { // inheritance
  public void run() {
  // overwriting method run() of class Thread
    System.out.println("hello from new thread");
  }
  public static void main (String args[]) {
    NewClass nc = new NewClass();
    nc.start();
  }
}
```

resumes execution usually before the new thread has terminated, i.e., the generating thread and the new thread are executed concurrently with each other.

The new thread is terminated when the execution of the `run()` method has been finished. This mechanism for thread creation is illustrated in Fig. 6.21 with a class `NewClass` whose `main()` method generates an object of `NewClass` and whose `run()` method is activated by calling the `start()` method of the newly created object. Thus, thread creation can be performed in two steps:

(1) definition of a class `NewClass` which inherits from `Thread` and which defines a `run()` method for the new thread;
(2) instantiation of an object `nc` of class `NewClass` and activation of `nc.start()`.

The creation method just described requires that the class `NewClass` inherits from `Thread`. Since Java does not support multiple inheritance, this method has the drawback that `NewClass` cannot be embedded into another inheritance hierarchy. Java provides interfaces to obtain a similar mechanism as multiple inheritance. For thread creation, the interface `Runnable` is used.

6.2.1.2 Using the Interface `Runnable`

The interface `Runnable` defines an abstract `run()` method as follows:

```
public interface Runnable {
  public abstract void run();
}
```

The predefined class `Thread` implements the interface `Runnable`. Therefore, each class which inherits from `Thread`, also implements the interface `Runnable`. Hence, instead of inheriting from `Thread` the newly defined class `NewClass` can directly implement the interface `Runnable`.

This way, objects of class `NewClass` are not thread objects. The creation of a new thread requires the generation of a new `Thread` object to which the object `NewClass` is passed as parameter. This is obtained by using the constructor

```
public Thread (Runnable target).
```

Using this constructor, the `start()` method of `Thread` activates the `run()` method of the `Runnable` object which has been passed as argument to the constructor.

This is obtained by the `run()` method of `Thread` which is specified as follows:

```
public void run() {
   if (target != null) target.run();
}
```

After activating `start()`, the `run()` method is executed by a separate thread which runs asynchronously with the calling thread. Thus, thread creation can be performed by the following steps:

(1) definition of a class `NewClass` which implements `Runnable` and which defines a `run()` method containing the code to be executed by the new thread;
(2) instantiation of a `Thread` object using the constructor `Thread (Runnable target)` and of an object of `NewClass` which is passed to the `Thread` constructor;
(3) activation of the `start()` method of the `Thread` object.

This is illustrated in Fig. 6.22 for a class `NewClass`. An object of this class is passed to the `Thread` constructor as parameter.

```
import java.lang.Thread;
public class NewClass implements Runnable {
  public void run() {
    System.out.println("hello from new thread");
  }
  public static void main (String args[]) {
    NewClass nc = new NewClass();
    Thread th = new Thread(nc);
    th.start(); // start() activates nc.run() in a new thread
  }
}
```

Fig. 6.22 Thread creation by using the interface `Runnable` based on the definition of a new class `NewClass`

6.2.1.3 Further Methods of the `Thread` Class

A Java thread can wait for the termination of another Java thread `t` by calling `t.join()`. This call blocks the calling thread until the execution of `t` is terminated. There are three variants of this method:

- `void join()`: the calling thread is blocked until the target thread is terminated;
- `void join (long timeout)`: the calling thread is blocked until the target thread is terminated or the given time interval `timeout` has passed; the time interval is given in milliseconds;
- `void join (long timeout, int nanos)`: the behavior is similar to `void join (long timeout)`; the additional parameter allows a more exact specification of the time interval using an additional specification in nanoseconds.

The calling thread will not be blocked if the target thread has not yet been started. The method

```
boolean isAlive()
```

of the `Thread` class gives information about the execution status of a thread: The method returns `true` if the target thread has been started but has not yet been terminated; otherwise, `false` is returned. The `join()` and `isAlive()` methods have no effect on the calling thread. A name can be assigned to a specific thread and can later be retrieved by using the methods

```
void setName (String name);
String getName();
```

An assigned name can later be used to identify the thread. A name can also be assigned at thread creation by using the constructor `Thread (String name)`. The `Thread` class defines static methods which affect the calling thread or provide information about program execution:

```
static Thread currentThread();
static void sleep (long milliseconds);
static void yield();
static int enumerate (Thread[] th_array);
static int activeCount();
```

Since these methods are static, they can be called without using a target `Thread` object. The call of `currentThread()` returns a reference to the `Thread` object of the calling thread. This reference can later be used to call non-static methods of the `Thread` object. The method `sleep()` blocks the execution of the calling thread until the specified time interval has passed; at this time, the thread again becomes ready for execution and can be assigned to an execution core or processor.

The method `yield()` is a directive to the Java Virtual Machine (JVM) to assign another thread with the same priority to the processor. If such a thread exists, then the scheduler of the JVM can bring this thread to execution. The use of `yield()` is useful for JVM implementations without a time-sliced scheduling, if threads perform long-running computations which do not block. The method `enumerate()` yields a list of all active threads of the program. The return value specifies the number of `Thread` objects collected in the parameter array `th_array`. The method `activeCount()` returns the number of active threads in the program. The method can be used to determine the required size of the parameter array before calling `enumerate()`.

Example Figure 6.23 gives an example of a class for performing a matrix multiplication with multiple threads. The input matrices are read into `in1` and `in2` by the main thread using the static method `ReadMatrix()`. The thread creation is performed by the constructor of the `MatMult` class such that each thread computes one row of the result matrix. The corresponding computations are specified in the `run()` method. All threads access the same matrices `in1`, `in2`, and `out` that have been allocated by the main thread. No synchronization is required, since each thread writes to a separate area of the result matrix `out`. □

6.2.2 Synchronization of Java Threads

The threads of a Java program access a shared address space. Suitable synchronization mechanisms have to be applied to avoid race conditions when a variable is accessed by several threads concurrently. Java provides `synchronized` blocks and methods to guarantee mutual exclusion for threads accessing shared data. A `synchronized` block or method avoids a concurrent execution of the block or method by two or more threads. A data structure can be protected by putting all accesses to it into `synchronized` blocks or methods, thus ensuring mutual exclusion. A synchronized increment operation of a counter can be realized by the following method `incr()`:

```
public class Counter {
  private int value = 0;
  public synchronized int incr() {
       value = value + 1;
       return value;
  }
}
```

Java implements the synchronization by assigning to each Java object an implicit mutex variable. This is achieved by providing the general class `Object` with an implicit mutex variable. Since each class is directly or indirectly derived from the class `Object`, each class inherits this implicit mutex variable, and every object

```
import java.lang.*;
import java.io.*;
class MatMult extends Thread {
  static int in1[][]; static int in2[][]; static int out[][];
  static int n=3; int row;
  MatMult (int i) {
    row=i;
    this.start();
  }
  public void run() {
  //compute a row of the result matrix
    int i,j;
    for(i=0;i<n;i++) {
      out[row][i]=0;
      for (j=0;j<n;j++)
        out[row][i]=out[row][i]+in1[row][j]*in2[j][i];
    }
  }
  public static void ReadMatrix (int in[][]) {
  //read the input matrix
    int i,j;
    BufferedReader br=new BufferedReader(new
      InputStreamReader(System.in));
    System.out.println("Enter the Matrix : ");
    for(i=0;i<n;i++)
      for(j=0;j<n;j++)
        try {
          in[i][j]=Integer.parseInt(br.readLine());
        } catch(Exception e){ }
  }
  public static void PrintMatrix (int out[][]) {
  //print the result matrix
    int i,j;
    System.out.println("OUTPUT :");
    for(i=0;i<n;i++)
      for(j=0;j<n;j++)
        System.out.println(out[i][j]);
  }
  public static void main(String args[]) {
    int i,j;
    in1=new int[n][n]; in2=new int[n][n];
    out=new int[n][n];
    ReadMatrix(in1); ReadMatrix(in2);
    MatMult mat[] = new MatMult[n];
    for(i=0;i<n;i++)
      mat[i]=new MatMult(i);
    try {
      for(i=0;i<n;i++)
        mat[i].join();
    } catch(Exception e){ }
    PrintMatrix(out);
  }
}
```

Fig. 6.23 Parallel matrix multiplication in Java

instantiated from any class implicitly possesses its own mutex variable. The activation of a `synchronized` method of an object `Ob` by a thread `t` has the following effects:

- When starting the `synchronized` method, `t` implicitly tries to lock the mutex variable of `Ob`. If the mutex variable is already locked by another thread `s`, thread `t` is blocked. The blocked thread becomes ready for execution again when the mutex variable is released by the locking thread `s`. The called `synchronized` method will only be executed after successfully locking the mutex variable of `Ob`.
- When `t` leaves the synchronized method called, it implicitly releases the mutex variable of `Ob` so that it can be locked by another thread.

A synchronized access to an object can be realized by declaring all methods accessing the object as `synchronized`. The object should only be accessed with these methods to guarantee mutual exclusion.

In addition to `synchronized` methods, Java provides `synchronized` blocks: Such a block is started with the keyword `synchronized` and the specification of an arbitrary object that is used for the synchronization in parenthesis. Instead of an arbitrary object, the synchronization is usually performed with the object whose method contains the `synchronized` block. The above method for the incrementation of a counter variable can be realized using a `synchronized` block as follows:

```
public int incr() {
  synchronized (this) {
       value = value + 1; return value;
  }
}
```

The synchronization mechanism of Java can be used for the realization of **fully synchronized objects** (also called **atomar objects**); these can be accessed by an arbitrary number of threads without any additional synchronization. To avoid race conditions, the synchronization has to be performed within the methods of the corresponding class of the objects. This class must have the following properties:

- all methods must be declared `synchronized`;
- no public entries are allowed that can be accessed without using a local method;
- all entries are consistently initialized by the constructors of the class;
- the objects remain in a consistent state also in case of exceptions.

Figure 6.24 demonstrates the concept of fully synchronized objects for the example of a class `ExpandableArray`; this is a simplified version of the predefined synchronized class `java.util.Vector`, see also [113]. The class implements an adaptable array of arbitrary objects, i.e., the size of the array can be increased or decreased according to the number of objects to be stored. The adaptation is realized by the method `add()`: If the array `data` is fully occupied when trying

Fig. 6.24 Example for a fully synchronized class

```
import java.lang.*;
import java.util.*;
public class ExpandableArray {
  private Object[] data;
  private int size = 0;
  public ExpandableArray(int cap) {
    data = new Object[cap];
  }
  public synchronized int size() {
    return size;
  }
  public synchronized Object get(int i)
    throws NoSuchElementException {
      if (i < 0 || i >= size)
        throw new NoSuchElementException();
      return data[i];
  }
  public synchronized void add(Object x) {
    if (size == data.length) { // array too small
      Object[] od = data;
      data = new Object[3 * (size + 1) / 2];
      System.arraycopy(od, 0, data, 0, od.length);
    }
    data[size++] = x;
  }
  public synchronized void removeLast()
    throws NoSuchElementException {
      if (size == 0)
        throw new NoSuchElementException();
      data[--size] = null;
  }
}
```

to add a new object, the size of the array will be increased by allocating a larger array and using the method `arraycopy()` from the `java.lang.System` class to copy the content of the old array into the new array. Without the synchronization included, the class cannot be used concurrently by more than one thread safely. A conflict could occur if, e.g., two threads tried to perform an `add()` operation at the same time.

6.2.2.1 Deadlocks

The use of fully synchronized classes avoids the occurrence of race conditions, but may lead to deadlocks when threads are synchronized with different objects. This is illustrated in Fig. 6.25 for a class `Account` which provides a method `swapBalance()` to swap account balances, see [113]. A deadlock can occur when `swapBalance()` is executed by two threads *A* and *B* concurrently: For two account objects `a` and `b`, if *A* calls `a.swapBalance(b)` and *B* calls `b.swap`

Fig. 6.25 Example for a deadlock situation

```
pulic class Account {
  private long balance;
  synchronized long getBalance() {return balance;}
  synchronized void setBalance(long v) {
    balance = v;
  }
  synchronized void swapBalance(Account other) {
    long t = getBalance();
    long v = other.getBalance();
    setBalance(v);
    other.setBalance(t);
  }
}
```

`Balance(a)` and *A* and *B* are executed on different processors or cores, a deadlock occurs with the following execution order:

- **time** T_1: thread *A* calls `a.swapBalance(b)` and locks the mutex variable of object `a`;
- **time** T_2: thread *A* calls `getBalance()` for object `a` and executes this function;
- **time** T_2: thread *B* calls `b.swapBalance(a)` and locks the mutex variable of object `b`;
- **time** T_3: thread *A* calls `b.getBalance()` and blocks because the mutex variable of `b` has previously been locked by thread *B*;
- **time** T_3: thread *B* calls `getBalance()` for object `b` and executes this function;
- **time** T_4: thread *B* calls `a.getBalance()` and blocks because the mutex variable of `a` has previously been locked by thread *A*.

The execution order is illustrated in Fig. 6.26. After time T_4, both threads are blocked: Thread *A* is blocked, since it could not acquire the mutex variable of object `b`. This mutex variable is owned by thread *B* and only *B* can free it. Thread *B* is blocked, since it could not acquire the mutex variable of object `a`. This mutex variable is owned by thread *A*, and only *A* can free it. Thus, both threads are blocked and none of them can proceed; a deadlock has occurred.

Deadlocks typically occur if different threads try to lock the mutex variables of the same objects in different orders. For the example in Fig. 6.25, thread *A* tries to lock first `a` and then `b`, whereas thread *B* tries to lock first `b` and then `a`. In this situation, a deadlock can be avoided by a backoff strategy or by using the same

Time	operation Thread *A*	operation Thread *B*	owner mutex `a`	owner mutex `b`
T_1	`a.swapBalance(b)`		*A*	–
T_2	`t = getBalance()`	`b.swapBalance(a)`	*A*	*B*
T_3	Blocked with respect to `b`	`t = getBalance()`	*A*	*B*
T_4		Blocked with respect to `a`	*A*	*B*

Fig. 6.26 Execution order to cause a deadlock situation for the class in Fig. 6.25

locking order for each thread, see also Sect. 6.1.2. A unique ordering of objects can be obtained by using the Java method `System.identityHashCode()` which refers to the default implementation `Object.hashCode()`, see [113]. But any other unique object ordering can also be used. Thus, we can give an alternative formulation of `swapBalance()` which avoids deadlocks, see Fig. 6.27. The new formulation also contains an alias check to ensure that the operation is only executed if different objects are used. The method `swapBalance()` is not declared `synchronized` any more.

```
public void swapBalance(Account other) {
  if (other == this) return;
  else if (System.identityHashCode(this) <
      System.identityHashCode(other))
    this.doSwap(other);
  else other.doSwap(this);
}
protected synchronized void doSwap(Account other) {
  long t = getBalance();
  long v = other.getBalance();
  setBalance(v);
  other.setBalance(t);
}
```

Fig. 6.27 Deadlock-free implementation of `swapBalance()` from Fig. 6.25

For the synchronization of Java methods, several issues should be considered to make the resulting programs efficient and safe:

- Synchronization is expensive. Therefore, `synchronized` methods should only be used for methods that can be called concurrently by several threads and that may manipulate common object data.
  If an application ensures that a method is always executed by a single thread at each point in time, then a synchronization can be avoided to increase efficiency.
- Synchronization should be restricted to critical regions to reduce the time interval of locking. For larger methods, the use of `synchronized` blocks instead of `synchronized` methods should be considered.
- To avoid unnecessary sequentializations, the mutex variable of the same object should not be used for the synchronization of different, non-contiguous critical sections.
- Several Java classes are internally synchronized; examples are `Hashtable`, `Vector`, and `StringBuffer`. No additional synchronization is required for objects of these classes.
- If an object requires synchronization, the object data should be put into `private` or `protected` instance fields to inhibit non-synchronized accesses from outside. All object methods accessing the instance fields should be declared as `synchronized`.
- For cases in which different threads access several objects in different orders, deadlocks can be prevented by using the same lock order for each thread.

6.2.2.2 Synchronization with Variable Lock Granularity

To illustrate the use of the synchronization mechanism of Java, we consider a synchronization class with a variable lock granularity, which has been adapted from [129].

The new class `MyMutex` allows the synchronization of arbitrary object accesses by explicitly acquiring and releasing objects of the class `MyMutex`, thus realizing a lock mechanism similar to mutex variables in Pthreads, see Sect. 6.1.2, p. 263. The new class also enables the synchronization of threads accessing different objects. The class `MyMutex` uses an instance field `OwnerThread` which indicates which thread has currently acquired the synchronization object. Figure 6.28 shows a first draft of the implementation of `MyMutex`.

The method `getMyMutex` can be used to acquire the explicit lock of the synchronization object for the calling thread. The lock is given to the calling thread by assigning `Thread.currentThread()` to the instance field `OwnerThread`. The `synchronized` method `freeMyMutex()` can be used to release a previously acquired explicit lock; this is implemented by assigning `null` to the instance field `OwnerThread`. If a synchronization object has already been locked by another thread, `getMyMutex()` repeatedly tries to acquire the explicit lock after a fixed time interval of 100 ms. The method `getMyMutex()` is not declared `synchronized`. The `synchronized` method `tryGetMyMutex()` is used to access the instance field `OwnerThread`. This protects the critical section for acquiring the explicit lock by using the implicit mutex variable of the synchronization object. This mutex variable is used for both `tryGetMyMutex()` and `freeMyMutex()`.

```
public class MyMutex {
  protected Thread OwnerThread = null;
  public void getMyMutex() {
    while (!tryGetMyMutex()) {
      try { Thread.sleep(100); }
      catch (InterruptedException e) { }
    }
  }
  public synchronized boolean tryGetMyMutex() {
    if (OwnerThread == null) {
      OwnerThread = Thread.currentThread();
      return true;
    }
    else return false;
  }
  public synchronized void freeMyMutex() {
    if (OwnerThread == Thread.currentThread())
      OwnerThread = null;
  }
}
```

Fig. 6.28 Synchronization class with variable lock granularity

Fig. 6.29 Implementation variant of `getMyMutex()`

```
public void getMyMutex() {
  for ( ; ; ) {
    synchronized(this) {
      if (OwnerThread == null) {
        OwnerThread = Thread.currentThread();
        break;
      }
    }
    try { Thread.sleep(100); }
    catch (InterruptedException e) { }
  }
}
```

Fig. 6.30 Implementation of a counter class with synchronization by an object of class `MyMutex`

```
public class Counter {
  private int value;
  private MyMutex flag = new MyMutex();
  public int incr() {
    int res;
    flag.getMyMutex();
    value = value + 1;
    res = value;
    flag.freeMyMutex();
    return res;
  }
}
```

If `getMyMutex()` had been declared `synchronized`, the activation of `getMyMutex()` by a thread T_1 would lock the implicit mutex variable of the synchronization object of the class `MyMutex` before entering the method. If another thread T_2 holds the explicit lock of the synchronization object, T_2 cannot release this lock with `freeMyMutex()` since this would require to lock the implicit mutex variable which is held by T_1. Thus, a deadlock would result. The use of an additional method `tryGetMyMutex()` can be avoided by using a `synchronized` block within `getMyMutex()`, see Fig. 6.29.

Objects of the new synchronization class `MyMutex` can be used for the explicit protection of critical sections. This can be illustrated for a counter class `Counter` to protect the counter manipulation, see Fig. 6.30.

6.2.2.3 Synchronization of Static Methods

The implementation of `synchronized` blocks and methods based on the implicit object mutex variables works for all methods that are activated with respect to an

```
public class MyStatic {
  public static synchronized void staticMethod(MyStatic obj) {
    // here, the class mutex is used
    obj.nonStaticMethod();
    synchronized(obj) {
      // here, additionally, the object mutex is used
    }
  }
  public synchronized void nonStaticMethod() {
    // using the object mutex
  }
}
```

Fig. 6.31 Synchronization of static methods

object. Static methods of a class are not activated with respect to an object and thus, there is no implicit object mutex variable. Nevertheless, static methods can also be declared `synchronized`. In this case, the synchronization is implemented by using the implicit mutex variable of the corresponding class object of the class `java.lang.Class` (`Class` mutex variable). An object of this class is automatically generated for each class defined in a Java program.

Thus, static and non-static methods of a class are synchronized by using different implicit mutex variables. A static `synchronized` method can acquire the mutex variable of the `Class` object and of an object of this class by using an object of this class for a `synchronized` block or by activating a `synchronized` non-static method for an object of this class. This is illustrated in Fig. 6.31 see [129]. Similarly, a `synchronized` non-static method can also acquire both the mutex variables of the object and of the `Class` object by calling a `synchronized` static method. For an arbitrary class `Cl`, the `Class` mutex variable can be directly used for a `synchronized` block by using

```
synchronized (Cl.class) {/*Code*/}
```

6.2.3 Wait and Notify

In some situations, it is useful for a thread to wait for an event or condition. As soon as the event occurs, the thread executes a predefined action. The thread waits as long as the event does not occur or the condition is not fulfilled. The event can be signaled by another thread; similarly, another thread can make the condition to be fulfilled. Pthreads provide condition variables for these situations. Java provides a similar mechanism via the methods `wait()` and `notify()` of the predefined `Object` class. These methods are available for each object of any class which is explicitly or

implicitly derived from the `Object` class. Both methods can only be used within `synchronized` blocks or methods. A typical usage pattern for `wait()` is

```
synchronized (lockObject) {
   while (!condition) { lockObject.wait(); }
   Action();
}
```

The call of `wait()` blocks the calling thread until another thread calls `notify()` for the same object. When a thread blocks by calling `wait()`, it releases the implicit mutex variable of the object used for the synchronization of the surrounding `synchronized` method or block. Thus, this mutex variable can be acquired by another thread.

Several threads may block waiting for the same object. Each object maintains a list of waiting threads. When another thread calls the `notify()` method of the same object, one of the waiting threads of this object is woken up and can continue running. Before resuming its execution, this thread first acquires the implicit mutex variable of the object. If this is successful, the thread performs the action specified in the program. If this is not successful, the thread blocks and waits until the implicit mutex variable is released by the owning thread by leaving a `synchronized` method or block.

The methods `wait()` and `notify()` work similarly as the operations `pthread_cond_wait()` and `pthread_cond_signal()` for condition variables in Pthreads, see Sect. 6.1.3, p. 270. The methods `wait()` and `notify()` are implemented using an implicit waiting queue for each object this waiting queue contains all blocked threads waiting to be woken up by a `notify()` operation. The waiting queue does not contain those threads that are blocked waiting for the implicit mutex variable of the object.

The Java language specification does not specify which of the threads in the waiting queue is woken up if `notify()` is called by another thread. The method `notifyAll()` can be used to wake up all threads in the waiting queue; this has a similar effect as `pthread_cond_broadcast()` in Pthreads. The method `notifyAll()` also has to be called in a `synchronized` block or method.

6.2.3.1 Producer–Consumer Pattern

The Java waiting and notification mechanism described above can be used for the implementation of a producer–consumer pattern using an item buffer of fixed size. Producer threads can put new items into the buffer and consumer threads can remove items from the buffer. Figure 6.32 shows a thread-safe implementation of such a buffer mechanism adapted from [113] using the `wait()` and `notify()` methods of Java. When creating an object of the class `BoundedBufferSignal`, an array `array` of a given size `capacity` is generated; this array is used as buffer.

Fig. 6.32 Realization of a thread-safe buffer mechanism using Java `wait()` and `notify()`

```
public class BoundedBufferSignal {
  private final Object[] array;
  private int putptr = 0;
  private int takeptr = 0;
  private int numel = 0; // number of items in buffer
  public BoundedBufferSignal (int capacity)
    throws IllegalArgumentException {
      if (capacity <= 0)
         throw new IllegalArgumentException();
      array = new Object[capacity];
  }
  public synchronized int size() {return numel; }
  public int capacity() {return array.length;}
  public synchronized void put(Object obj)
    throws InterruptedException {
    while (numel == array.length)
      wait(); // buffer full
    array [putptr] = obj;
    putptr = (putptr +1) % array.length;
    if (numel++ == 0)
      notifyAll(); // wake up all threads
  }
  public synchronized Object take()
    throws InterruptedException {
    while (numel == 0)
      wait(); // buffer empty
    Object x = array [takeptr];
    takeptr = (takeptr +1) % array.length;
    if (numel-- == array.length)
      notifyAll(); // wake up all threads
    return x;
  }
}
```

The class provides a `put()` method to enable a producer to enter an item into the buffer and a `take()` method to enable a consumer to remove an item from the buffer. A buffer object can have one of three states: full, partially full, and empty. Figure 6.33 illustrates the possible transitions between the states when calling `take()` or `put()`. The states are characterized by the following conditions:

State	**Condition**	Put possible	Take possible
Full	`size == capacity`	No	Yes
Partially full	`0 < size < capacity`	Yes	Yes
Empty	`size == 0`	Yes	No

If the buffer is full, the execution of the `put()` method by a producer thread will block the executing thread; this is implemented by a `wait()` operation. If the `put()` method is executed for a previously empty buffer, all waiting (consumer)

Fig. 6.33 Illustration of the states of a thread-safe buffer mechanism

threads will be woken up using `notifyAll()` after the item has been entered into the buffer. If the buffer is empty, the execution of the `take()` method by a consumer thread will block the executing thread using `wait()`. If the `take()` method is executed for a previously full buffer, all waiting (producer) threads will be woken up using `notifyAll()` after the item has been removed from the buffer. The implementation of `put()` and `take()` ensures that each object of the class `BoundedBufferSignal` can be accessed concurrently by an arbitrary number of threads without race conditions.

6.2.3.2 Modification of the `MyMutex` Class

The methods `wait()` and `notify()` can be used to improve the synchronization class `MyMutex` from Fig. 6.28 by avoiding the active waiting in the method `getMyMutex()`, see Fig. 6.34 (according to [129]).

```
public synchronized void getMyMutex() {
  while (!tryGetMyMutex()) {
    try { wait(); }
    catch (InterruptedException e) { }
  }
}
public synchronized boolean tryGetMyMutex() {
  if (OwnerThread == null) {
    OwnerThread = Thread.currentThread() ;
    lockCount = 1; return true;
  }
  if (OwnerThread = Thread.currentThread()){
    lockCount ++; return true;
  }
  return false;
}
public synchronized Thread getMutexOwner() {
  return OwnerThread;
}
public synchronized void freeMyMutex() {
  if (OwnerThread == Thread.currentThread()) {
    if (--lockCount == 0) {
      OwnerThread = null;
      notify();
    }
  }
}
```

Fig. 6.34 Realization of the synchronization class `MyMutex` with `wait()` and `notify()` avoiding active waiting

Additionally, the modified implementation realizes a nested locking mechanism which allows multiple locking of a synchronization object by the same thread. The number of locks is counted in the variable `lockCount`; this variable is initialized to 0 and is incremented or decremented by each call of `getMyMutex()` or `freeMyMutex()`, respectively. In Fig. 6.34, the method `getMyMutex()` is now also declared `synchronized`. With the implementation in Fig. 6.28, this would lead to a deadlock. But in Fig. 6.34, no deadlock can occur, since the activation of `wait()` releases the implicit mutex variable before the thread is suspended and inserted into the waiting queue of the object.

6.2.3.3 Barrier Synchronization

A barrier synchronization is a synchronization point at which each thread waits until all participating threads have reached this synchronization point. Only then the threads proceed with their execution. A barrier synchronization can be implemented in Java using `wait()` and `notify()`. This is shown in Fig. 6.35 for a class `Barrier`, see also [129]. The `Barrier` class contains a constructor which initializes a `Barrier` object with the number of threads to wait for (`t2w4`). The actual synchronization is provided by the method `waitForRest()`. This method must be called by each thread at the intended synchronization point. Within the

```
public class Barrier() {
  private int t2w4;
  private InterruptedException e;
  public Barrier(int n) {
    this.t2w4 = n;
  }
  public synchronized int waitForRest()
      throws InterruptedException {
    int nThreads = --t2w4;
    if (e != null) throw e;
    if (t2w4 <= 0) {
      notifyAll(); return nThreads;
    }
    while (t2w4 > 0) {
      if (e != null) throw e;
      try { wait(); }
      catch(InterruptedException e) { this.e = e; notifyAll(); }
    }
    return nThreads;
  }
}
```

Fig. 6.35 Realization of a barrier synchronization in Java with `wait()` and `notify()`

Fig. 6.36 Use of the `Barrier` class for the realization of a multi-phase algorithm

```
public class ProcessIt implements Runnable {
  String is[];
  Barrier bpStart, bp1, bp2, bpEnd;
  public ProcessIt(String sources[]) {
    is = sources;
    bpStart = new Barrier(sources.length);
    bp1 = new Barrier(sources.length);
    bp2 = new Barrier(sources.length);
    bpEnd = new Barrier(sources.length);
    for (int i = 0; i < sources.length; i++)
      new Thread(this).start();
  }
  public void run() {
    try {
      int i = bpStart.waitForRest();
      doPhase1(is[i]);
      bp1.waitForRest();
      doPhase2(is[i]);
      bp2.waitForRest();
      doPhase3(is[i]);
      bpEnd.waitForRest();
    }
    catch (InterruptedException e)
  }
  public static void main(String args[]) {
    ProcessIt pi = new ProcessIt(args);
  }
}
```

method, each thread decrements `t2w4` and calls `wait()` if `t2w4` is > 0. This blocks each arriving thread within the `Barrier` object. The last arriving thread wakes up all waiting threads using `notifyAll()`.

Objects of the `Barrier` class can be used only once, since the synchronization counter `t2w4` is decremented to 0 during the synchronization process. An example for the use of the `Barrier` class for the synchronization of a multi-phase computation is given in Fig. 6.36, see also [129]. The program illustrates an algorithm with three phases (`doPhase1()`, `doPhase2()`, `doPhase3()`) which are separated from each other by a barrier synchronization using `Barrier` objects `bp1`, `bp2`, and `bpEnd`. Each of the threads created in the constructor of `ProcessIt` executes the three phases.

6.2.3.4 Condition Variables

The mechanism provided by `wait()` and `notify()` in Java has some similarities to the synchronization mechanism of condition variables in Pthreads, see Sect. 6.1.3,

p. 270. The main difference lies in the fact that `wait()` and `notify()` are provided by the general `Object` class. Thus, the mechanism is implicitly bound to the internal mutex variable of the object for which `wait()` and `notify()` are activated. This facilitates the use of this mechanism by avoiding the explicit association of a mutex variable as needed when using the corresponding mechanism in Pthreads. But the fixed binding of `wait()` and `notify()` to a specific mutex variable also reduces the flexibility, since it is not possible to combine an arbitrary mutex variable with the waiting queue of an object.

When calling `wait()` or `notify()`, a Java thread must be the owner of the mutex variable of the corresponding object; otherwise an exception `IllegalMonitorStateException` is raised. With the mechanism of `wait()` and `notify()` it is not possible to use the same mutex variable for the synchronization of the waiting queues of different objects. This would be useful, e.g., for the implementation of producer and consumer threads with a common data buffer, see, e.g., Fig. 6.18. But `wait()` and `notify()` can be used for the realization of a new class which mimics the mechanism of condition variables in Pthreads. Figure 6.37 shows an implementation of such a class `CondVar`, see also [129, 113]. The class `CondVar` provides the methods `cvWait()`, `cvSignal()`, and `cvBroadcast()` which mimic the behavior of `pthread_cond_wait()`, `pthread_cond_signal()`, and `pthread_cond_broadcast()`, respectively. These methods allow the use of an arbitrary mutex variable for the synchronization. This mutex variable is provided as a parameter of type `MyMutex` for each of the methods, see Fig. 6.37.

Thus a single mutex variable of type `MyMutex` can be used for the synchronization of several condition variables of type `CondVar`. When calling `cvWait()`, a thread will be blocked and put in the waiting queue of the corresponding object of type `CondVar`. The internal synchronization within `cvWait()` is performed with the internal mutex variable of this object. The class `CondVar` also allows a simple porting of Pthreads programs with condition variables to Java programs.

Figure 6.38 shows as example the realization of a buffer mechanism with producer and consumer threads by using the new class `CondVar`, see also [113]. A producer thread can insert objects into the buffer by using the method `put()`. A consumer thread can remove objects from the buffer by using the method `take()`. The condition objects `notFull` and `notEmpty` of type `CondVar` use the same mutex variable `mutex` for synchronization.

6.2.4 Extended Synchronization Patterns

The synchronization mechanisms provided by Java can be used to implement more complex synchronization patterns which can then be used in parallel application programs. This will be demonstrated in the following for the example of a semaphore mechanism, see p. 138.

```
public class CondVar {
  private MyMutex syncVar; /* use MyMutex for synchronization */
  public CondVar() {
    this(new MyMutex());
  }
  public CondVar(MyMutex sv) {
    syncVar = sv;
  }
  public void cvWait() throws InterruptedException {
    cvWait(syncVar, 0);
  }
  public void cvWait(MyMutex sv) throws InterruptedException {
    cvWait(sv, 0);
  }
  public void cvWait(int millis) throws InterruptedException {
    cvWait(syncVar, millis);
  }
  public void cvWait(MyMutex sv, int millis)
      throws InterruptedException {
    int i = 0;
    InterruptedException exception;
    synchronized (this) {
      if (sv.getMutexOwner() != Thread.currentThread())
        throw new IllegalMonitorStateException ("thread not owner");
      while (sv.getMutexOwner() == Thread.currentThread()) {
        i++; sv.freeMyMutex();
      }
      try { if (millis == 0) wait(); else wait(millis); }
      catch (InterruptedException e) { exception = e; }
    }
    for (; i > 0; i--) sv.getMyMutex();
    if (exception != null) throw exception;
  }
  public void cvSignal() {
    cvSignal(syncVar);
  }
  public synchronized void cvSignal(MyMutex sv) {
    if (sv.getMutexOwner() != Thread.currentThread())
      throw new IllegalMonitorStateException ("thread not owner");
    notify();
  }
  public void cvBroadcast() {
    cvBroadcast(syncVar);
  }
  public synchronized void cvBroadcast(MyMutex sv) {
    if (sv.getMutexOwner() != Thread.currentThread())
      throw new IllegalMonitorStateException("thread not owner");
    notifyAll();
  }
}
```

Fig. 6.37 Class `CondVar` for the realization of the Pthreads condition variable mechanism using the Java signaling mechanism

```
class PThreadsStyleBuffer {
   private final MyMutex mutex = new MyMutex();
   private final CondVar notFull = new CondVar(mutex);
   private final CondVar notEmpty = new CondVar(mutex);
   private int count = 0;
   private int takePtr = 0;
   private int putPtr = 0;
   private final Object[] array;

   public PThreadsStyleBuffer(int capacity) {
      array = new Object[capacity];
   }

   public void put(Object x) throws InterruptedException {
      mutex.getMyMutex();
      try {
          while (count == array.length)
             notFull.cvWait();

          array[putPtr] = x;
          putPtr = (putPtr + 1) % array.length;
          ++count;
          notEmpty.cvSignal();
      }
      finally {
          mutex.freeMyMutex();
      }
   }

   public Object take() throws InterruptedException {
      Object x = null;
      mutex.getMyMutex();
      try {
          while (count == 0)
             notEmpty.cvWait();

          x = array[takePtr];
          array[takePtr] = null;
          takePtr = (takePtr + 1) % array.length;
          --count;
          notFull.cvSignal();
      }
      finally {
          mutex.freeMyMutex();
      }
      return x;
   }
}
```

Fig. 6.38 Implementation of a buffer mechanism for producer and consumer threads

Fig. 6.39 Implementation of a semaphore mechanism

```
public class Semaphore {
  private long counter;
  public Semaphore(long init) {
    counter = init;
  }
  public void acquire()
      throws InterruptedException {
    if (Thread.interrupted())
      throw new InterruptedException();
    synchronized (this) {
      try {
         while (counter <= 0) wait();
         counter--;
      }
      catch (InterruptedException ie) {
         notify(); throw ie;
      }
    }
  }
  public synchronized void release() {
    counter++;
    notify();
  }
}
```

A semaphore mechanism can be implemented in Java by using `wait()` and `notify()`. Figure 6.39 shows a simple implementation, see also [113, 129]. The method `acquire()` waits (if necessary), until the internal counter of the semaphore object has reached at least the value 1. As soon as this is the case, the counter is decremented. The method `release()` increments the counter and uses `notify()` to wake up a waiting thread that has been blocked in `acquire()` by calling `wait()`. A waiting thread can only exist, if the counter had the value 0 before incrementing it. Only in this case, it can be blocked in `acquire()`. Since the counter is only incremented by one, it is sufficient to wake up a single waiting thread. An alternative would be to use `notifyAll()`, which wakes up all waiting threads. Only one of these threads would succeed in decrementing the counter, which would then have the value 0 again. Thus, all other threads that had been woken up would be blocked again by calling `wait()`.

The semaphore mechanism shown in Fig. 6.39 can be used for the synchronization of producer and consumer threads. A similar mechanism has already been implemented in Fig. 6.32 by using `wait()` and `notify()` directly. Figure 6.41 shows an alternative implementation with semaphores, see [113]. The producer

Fig. 6.40 Class `BufferArray` for buffer management

```
public class BufferArray {
  private final Object[] array;
  private int putptr = 0;
  private int takeptr = 0;
  public BufferArray (int n) {
    array = new Object[n];
  }
  public synchronized void insert (Object obj) {
    array[putptr] = obj;
    putptr = (putptr +1) % array.length;
  }
  public synchronized Object extract() {
    Object x = array[takeptr];
    array[takeptr] = null;
    takeptr = (takeptr +1) % array.length;
    return x;
  }
}
```

stores the objects generated into a buffer of fixed size, the consumer retrieves objects from this buffer for further processing. The producer can only store objects in the buffer, if the buffer is not full. The consumer can only retrieve objects from the buffer, if the buffer is not empty. The actual buffer management is done by a separate class `BufferArray` which provides methods `insert()` and `extract()` to insert and retrieve objects, see Fig. 6.40. Both methods are `synchronized`, so multiple threads can access objects of this class without conflicts. The class `BufferArray` does not provide a mechanism to control buffer overflow.

The class `BoundedBufferSema` in Fig. 6.41 provides the methods `put()` and `take()` to store and retrieve objects in a buffer. Two semaphores `putPermits` and `takePermits` are used to control the buffer management. At each point in time, these semaphores count the number of permits to store (producer) and retrieve (consumer) objects. The semaphore `putPermits` is initialized to the buffer size, the semaphore `takePermits` is initialized to 0. When storing an object by using `put()`, the semaphore `putPermits` is decremented with `acquire()`; if the buffer is full, the calling thread is blocked when doing this. After an object has been stored in the buffer with `insert()`, a waiting consumer thread (if present) is woken up by calling `release()` for the semaphore `takePermits`. Retrieving an object with `take()` works similarly with the role of the semaphores exchanged.

In comparison to the implementation in Fig. 6.32, the new implementation in Fig. 6.41 uses two separate objects (of type `Semaphore`) for buffer control. Depending on the specific situation, this can lead to a reduction of the synchronization overhead: In the implementation from Fig. 6.32 *all* waiting threads are woken up in `put()` and `take()`. But only one of these can proceed and retrieve an object from the buffer (consumer) or store an object into the buffer (producer). All other threads are blocked again. In the implementation from Fig. 6.41, only one thread is woken up.

Fig. 6.41 Buffer management with semaphores

```
pulic class BoundedBufferSema {
  private final BufferArray buff;
  private final Semaphore putPermits;
  private final Semaphore takePermits;
  public BoundedBufferSema(int capacity)
    throws IllegalArgumentException {
        if (capacity <= 0)
          throw new IllegalArgumentException();
        buff = new BufferArray(capacity);
        putPermits = new Semaphore(capacity);
        takePermits = new Semaphore(0);
  }
  public void put(Object x)
        throws InterruptedException {
    putPermits.acquire();
    buff.insert(x);
    takePermits.release();
  }
  public Object take()
        throws InterruptedException {
    takePermits.acquire();
    Object x = buff.extract();
    putPermits.release();
    return x;
  }
}
```

6.2.5 Thread Scheduling in Java

A Java program may consist of several threads which can be executed on one or several of the processors of the execution platform. The threads which are ready for execution compete for execution on a free processor. The programmer can influence the mapping of threads to processors by assigning priorities to the threads. The minimum, maximum, and default priorities for Java threads are specified in the following fields of the `Thread` class:

```
public static final int MIN_PRIORITY  // normally 1
public static final int MAX_PRIORITY  // normally 10
public static final int NORM_PRIORITY // normally 5
```

A *large* priority value corresponds to a *high* priority. The thread which executes the `main()` method of a class has by default the priority `Thread.NORM_PRIORITY`. A newly created thread has by default the same priority as the generating thread. The current priority of a thread can be retrieved or dynamically changed by using the methods

```
public int getPriority();
public int setPriority(int prio);
```

of the `Thread` class. If there are more executable threads than free processors, a thread with a larger priority is usually favored by the scheduler of the JVM. The exact mechanism for selecting a thread for execution may depend on the implementation of a specific JVM. The Java specification does not define an exact scheduling mechanism to increase flexibility for the implementation of the JVM on different operating systems and different execution platforms. For example, the scheduler might always bring the thread with the largest priority to execution, but it could also integrate an aging mechanism to ensure that threads with a lower priority will be mapped to a processor from time to time to avoid starvation and implement fairness.

Since there is no exact specification for the scheduling of threads with different priorities, priorities cannot be used to replace synchronization mechanisms. Instead, priorities can only be used to express the relative importance of different threads to bring the most important thread to execution in case of doubt.

When using threads with different priorities, the problem of **priority inversion** can occur, see also Sect. 6.1.11, p. 303. A priority inversion happens if a thread with a high priority is blocked to wait for a thread with a low priority, e.g., because this thread has locked the same mutex variable that the thread with the high priority tries to lock. The thread with a low priority can be inhibited from proceeding its execution and releasing the mutex variable as soon as a thread with a medium priority is ready for execution. In this constellation, the thread with high priority can be prevented from execution in favor of the thread with a medium priority.

The problem of priority inversion can be avoided by using **priority inheritance**, see also Sect. 6.1.11: If a thread with high priority is blocked, e.g., because of an activation of a `synchronized` method, then the priority of the thread that currently controls the critical synchronization object will be increased to the high priority of the blocked thread. Then, no thread with medium priority can inhibit the thread with high priority from execution. Many JVMs use this method, but this is not guaranteed by to the Java specification.

6.2.6 Package `java.util.concurrent`

The `java.util.concurrent` package provides additional synchronization mechanisms and classes which are based on the standard synchronization mechanisms described in the previous section, like `synchronized` blocks, `wait()` and `notify()`. The package is available for Java platforms starting with the Java2 platform (Java2 Standard Edition 5.0, J2SE 5.0).

The additional mechanisms provide more abstract and flexible synchronization operations, including atomic variables, lock variables, barrier synchronization, condition variables, and semaphores, as well as different thread-safe data structures like queues, hash-maps, or array lists. The additional classes are similar to those

described in [113]. In the following, we give a short overview of the package and refer to [70] for a more detailed description.

6.2.6.1 Semaphore Mechanism

The class `Semaphore` provides an implementation of a counting semaphore, which is similar to the mechanism given in Fig. 6.17. Internally, a `Semaphore` object maintains a counter which counts the number of permits. The most important methods of the `Semaphore` class are

```
void acquire();
void release();
boolean tryAcquire();
boolean tryAcquire(int permits, long timeout,
                   TimeUnit unit)
```

The method `acquire()` asks for a permit and blocks the calling thread if no permit is available. If a permit is currently available, the internal counter for the number of available permits is decremented and control is returned to the calling thread.

The method `release()` adds a permit to the semaphore by incrementing the internal counter. If another thread is waiting for a permit of this semaphore, this thread is woken up. The method `tryAcquire()` asks for a permit to a semaphore object. If a permit is available, a permit is acquired by the calling thread and control is returned immediately with return value `true`. If no permit is available, control is also returned immediately, but with return value `false`; thus, in contrast to `acquire()`, the calling thread is not blocked. There exist different variants of the method `tryAcquire()` with varying parameters allowing the additional specification of a number of permits to acquire (parameter `permits`), a waiting time (parameter `timeout`) after which the attempt of acquiring the specified number of permits is given up with return value `false`, as well as a time unit (parameter `unit`) for the waiting time. If not enough permits are available when calling a timed `tryAcquire()`, the calling thread is blocked until one of the following events occurs:

- the number of requested permits becomes available because other threads call `release()` for this semaphore; in this case, control is returned to the calling thread with return value `true`;
- the specified waiting time elapses; in this case, control is returned with return value `false`; no permit is acquired in this case, also if some of the requested permits would have been available.

6.2.6.2 Barrier Synchronization

The class `CyclicBarrier` provides an implementation of a barrier synchronization. The prefix *cyclic* refers to the fact that an object of this class can be re-used

again after all participating threads have passed the barrier. The constructors of the class

```
public CyclicBarrier (int n);
public CyclicBarrier (int n, Runnable action);
```

allow the specification of a number `n` of threads that must pass the barrier before execution continues after the barrier. The second constructor allows the additional specification of an operation `action` that is executed as soon as all threads have passed the barrier. The most important methods of `CyclicBarrier` are `await()` and `reset()`. By calling `await()` a thread waits at the barrier until the specified number of threads have reached the barrier. A barrier object can be reset into its original state by calling `reset()`.

6.2.6.3 Lock Mechanisms

The package `java.util.concurrent.locks` contains interfaces and classes for locks and for waiting for the occurrence of conditions. The interface `Lock` defines locking mechanisms which go beyond the standard `synchronized` methods and blocks and are not limited to the synchronization with the implicit mutex variables of the objects used. The most important methods of `Lock` are

```
void lock();
boolean tryLock();
boolean tryLock(long time, TimeUnit unit);
void unlock();
```

The method `lock()` tries to lock the corresponding lock object. If the lock has already been set by another thread, the executing thread is blocked until the locking thread releases the lock by calling `unlock()`. If the lock object has not been set by another thread when calling `lock()`, the executing thread becomes the owner of the lock without waiting.

The method `tryLock()` also tries to lock a lock object. If this is successful, the return value is `true`. If the lock object is already set by another thread, the return value is `false`; in contrast to `lock()`, the calling thread is not blocked in this case. For the method `tryLock()`, additional parameters can be specified to set a waiting time after which control is resumed also if the lock is not available, see `tryAcquire()` of the class `Semaphore`. The method `unlock()` releases a lock which has previously been set by the calling thread.

The class `ReentrantLock()` provides an implementation of the interface `Lock`. The constructors of this class

```
public ReentrantLock();
public ReentrantLock(boolean fairness);
```

```
import java.util.concurrent.locks.*;
pulic class NewClass {
  private ReentrantLock lock = new ReentrantLock();
  //...
  public void method() {
    lock.lock();
    try {
        //...
    } finally { lock.unlock(); }
  }
}
```

Fig. 6.42 Illustration of the use of `ReentrantLock` objects

allow the specification of an additional fairness parameter `fairness`. If this is set to `true`, the thread with the longest waiting time can access the lock object if several threads are waiting concurrently for the same lock object. If the fairness parameter is not used, no specific access order can be assumed. Using the fairness parameter can lead to an additional management overhead and hence to a reduced throughput. A typical usage of the class `ReentrantLock` is illustrated in Fig. 6.42.

6.2.6.4 Signal Mechanism

The interface `Condition` from the package `java.util.concurrent.locks` defines a signal mechanism with condition variables which allows a thread to wait for a specific condition. The occurrence of this condition is shown by a signal of another thread, similar to the functionality of condition variables in Pthreads, see Sect. 6.1.3, p. 270. A condition variable is always bound to a lock object, see interface `Lock`. A condition variable to a lock object can be created by calling the method

```
Condition newCondition().
```

This method is provided by all classes which implement the interface `Lock`. The condition variable returned by the method is bound to the lock object for which the method `newCondition()` has been called. For condition variables, the following methods are available:

```
void await();
void await(long time, TimeUnit unit);
void signal();
void signalAll();
```

The method `await()` blocks the executing thread until it is woken up by another thread by `signal()`. Before blocking, the executing thread releases the lock object as an atomic operation. Thus, the executing thread has to be the owner of the lock object before calling `await()`. After the blocked thread is woken up

```
import java.util.concurrent.locks.*;
pulic class BoundedBufferCondition {
  private Lock lock = new ReentrantLock();
  private Condition notFull = lock.newCondition();
  private Condition notEmpty = lock.newCondition();
  private Object[] items = new Object[100];
  private int putptr=0, takeptr=0, count=0;
  public void put (Object x)
       throws InterruptedException {
    lock.lock();
    try {
       while (count == items.length)
         notFull.await();
       items[putptr] = x;
       putptr = (putptr +1) % items.length;
       ++count;
       notEmpty.signal();
    } finally { lock.unlock(); }
  }
  public Object take()
       throws InterruptedException {
    lock.lock();
    try {
       while (count == 0)
         notEmpty.await();
       Object x = items[takeptr];
       takeptr = (takeptr +1) % items.length;
       --count;
       notFull.signal();
       return x;
    } finally {lock.unlock();}
  }
}
```

Fig. 6.43 Realization of a buffer mechanism by using condition variables

again by a `signal()` of another thread, it first must try to set the lock object again. Only after this is successful, the thread can proceed with its computations.

There is a variant of `await()` which allows the additional specification of a waiting time. If this variant is used, the calling thread is woken up after the time interval has elapsed and if no `signal()` of another thread has arrived in the meantime. By calling `signal()`, a thread can wake up another thread which is waiting for a condition variable. By calling `signalAll()`, *all* waiting threads of the condition variable are woken up. The use of condition variables for the realization of a buffer mechanism is illustrated in Fig. 6.43, see [70]. The condition variables are used in a similar way as the semaphore objects in Fig. 6.41.

6.2.6.5 Atomic Operations

The package `java.util.concurrent.atomic` provides atomic operations for simple data types, allowing a lock-free access to single variables. An example is the class `AtomicInteger` which comprises the following methods:

```
boolean compareAndSet (int expect, int update);
int getAndIncrement();
```

The first method sets the value of the variable to the value `update`, if the variable previously had the value `expect`. In this case, the return value is `true`. If the variable does not have the expected value, the return value is `false`; no operation is performed. The operation is performed *atomically*, i.e., during the execution, the operation cannot be interrupted.

The second method increments the value of the variable atomically and returns the previous value of the variable as a result. The class `AtomicInteger` provides plenty of similar methods.

6.2.6.6 Task-Based Execution of Programs

The package `java.util.concurrent` also provides a mechanism for a task-based formulation of programs. A task is a sequence of operations of the program which can be executed by an arbitrary thread. The execution of tasks is supported by the interface `Executor`:

```
public interface Executor {
   void execute (Runnable command);
}
```

where `command` is the task which is brought to execution by calling `execute()`. A simple implementation of the method `execute()` might merely activate the method `command.run()` in the current thread. More sophisticated implementations may queue `command` for execution by one of a set of threads. For multicore processors, several threads are typically available for the execution of tasks. These threads can be combined in a thread pool where each thread of the pool can execute an arbitrary task.

Compared to the execution of each task by a separate thread, the use of task pools typically leads to a smaller management overhead, particularly if the tasks consist of only a few operations. For the organization of thread pools, the class `Executors` can be used. This class provides methods for the generation and management of thread pools. Important methods are

```
static ExecutorService newFixedThreadPool(int n);
static ExecutorService newCachedThreadPool();
static ExecutorService newSingleThreadExecutor();
```

The first method generates a thread pool which creates new threads when executing tasks until the maximum number `n` of threads has been reached. The second method generates a thread pool for which the number of threads is dynamically adapted to the number of tasks to be executed. Threads are terminated if they are not used for a specific amount of time (60 s). The third method generates a single thread which executes a set of tasks. To support the execution of task-based programs the

interface `ExecutorService` is provided. This interface inherits from the interface `Executor` and comprises methods for the termination of thread pools. The most important methods are

```
void shutdown();
List<Runnable> shutdownNow();
```

The method `shutdown()` has the effect that the thread pool does not accept further tasks for execution. Tasks which have already been submitted are still executed before the shutdown. In contrast, the method `shutdownNow()` additionally stops the tasks which are currently in execution; the execution of waiting tasks is not started. The set of waiting tasks is provided in the form of a list as return value. The class `ThreadPoolExecutor` is an implementation of the interface `ExecutorService`.

```
import java.io.IOException;
import java.net.*;
import java.util.concurrent.*;

pulic class TaskWebServer {
  static class RunTask implements Runnable {
    private Socket myconnection;
    public RunTask (Socket connection) {
       myconnection = connection;
    }
    public void run() {
       // handleRequest(myconnection);
    }
  }
  public static void main (String[] args)
       throws IOException {
    ServerSocket s = new ServerSocket(80);
    ExecutorService pool =
       Executors.newFixedThreadPool(10);
    try {
       while (true) {
         Socket connection = s.accept();
         Runnable task = new RunTask(connection)
         pool.execute(task);
       }
    } catch (IOException ex) {
       pool.shutdown();
    }
  }
}
```

Fig. 6.44 Draft of a task-based web server

Figure 6.44 illustrates the use of a thread pool for the realization of a web server, see [70], which waits for connection requests of clients at a `ServerSocket` object. If a client request arrives, it is computed as a separate task by submitting this task with `execute()` to a thread pool. Each task is generated as a `Runnable` object. The operation `handleRequest()` to be executed for the request is specified as `run()` method. The maximum size of the thread pool is set to 10.

6.3 OpenMP

OpenMP is a portable standard for the programming of shared memory systems. The OpenMP API (application program interface) provides a collection of compiler directives, library routines, and environmental variables. The compiler directives can be used to extend the sequential languages Fortran, C, and C++ with single program multiple data (SPMD) constructs, tasking constructs, work-sharing constructs, and synchronization constructs. The use of shared and private data is supported. The library routines and the environmental variable control the runtime system.

The OpenMP standard was designed in 1997 and is owned and maintained by the OpenMP Architecture Review Board (ARB). Since then many vendors have included the OpenMP standard in their compilers. Currently most compilers support Version 2.5 from May 2005 [131]. The most recent update is Version 3.0 from May 2008 [132]. Information about OpenMP and the standard definition can be found at the following web site: `http://www.openmp.org`.

The programming model of OpenMP is based on cooperating **threads** running simultaneously on multiple processors or cores. Threads are created and destroyed in a **fork–join** pattern. The execution of an OpenMP program begins with a single thread, the initial thread, which executes the program sequentially until a first `parallel` construct is encountered. At the parallel construct the initial thread creates a team of threads consisting of a certain number of new threads and the initial thread itself. The initial thread becomes the master thread of the team. This fork operation is performed implicitly. The program code inside the parallel construct is called a **parallel region** and is executed in parallel by all threads of the team. The parallel execution mode can be an SPMD style; but an assignment of different tasks to different threads is also possible. OpenMP provides directives for different execution modes, which will be described below. At the end of a parallel region there is an implicit barrier synchronization, and only the master thread continues its execution after this region (implicit join operation). Parallel regions can be nested and each thread encountering a parallel construct creates a team of threads as described above.

The memory model of OpenMP distinguishes between shared memory and private memory. All OpenMP threads of a program have access to the same shared memory. To avoid conflicts, race conditions, or deadlocks, synchronization mechanisms have to be employed, for which the OpenMP standard provides appropri-

ate library routines. In addition to shared variables, the threads can also use private variables in the *threadprivate* memory, which cannot be accessed by other threads.

An OpenMP program needs to include the header file `<omp.h>`. The compilation with appropriate options translates the OpenMP source code into multithreaded code. This is supported by several compilers. The Version 4.2 of GCC and newer versions support OpenMP; the option `-fopenmp` has to be used. Intel's C++ compiler Version 8 and newer versions also support the OpenMP standard and provide additional Intel-specific directives. A compiler supporting OpenMP defines the variable `_OPENMP` if the OpenMP option is activated.

An OpenMP program can also be compiled into sequential code by a translation without the OpenMP option. The translation ignores all OpenMP directives. However, for the translation into correct sequential code special care has to be taken for some OpenMP runtime functions. The variable `_OPENMP` can be used to control the translation into sequential or parallel code.

6.3.1 Compiler Directives

In OpenMP, parallelism is controlled by compiler directives. For C and C++, OpenMP directives are specified with the `#pragma` mechanism of the C and C++ standards. The general form of an OpenMP directive is

```
#pragma omp directive [clauses [ ] ...]
```

written in a single line. The clauses are optional and are different for different directives. Clauses are used to influence the behavior of a directive. In C and C++, the directives are case sensitive and apply only to the next code line or to the block of code (written within brackets { and }) immediately following the directive.

6.3.1.1 Parallel Region

The most important directive is the `parallel` construct mentioned before with syntax

```
#pragma omp parallel [clause [clause] ... ]
{ // structured block ... }
```

The parallel construct is used to specify a program part that should be executed in parallel. Such a program part is called a *parallel region*. A team of threads is created to execute the parallel region in parallel. Each thread of the team is assigned a unique thread number, starting from zero for the master thread up to the number of threads minus one. The parallel construct ensures the creation of the team but does not distribute the work of the parallel region among the threads of the team. If there

is no further explicit distribution of work (which can be done by other directives), all threads of the team execute the same code on possibly different data in an SPMD mode. One usual way to execute on different data is to employ the thread number also called *thread id*. The user-level library routine

```
int omp_get_thread_num()
```

returns the thread id of the calling thread as integer value. The number of threads remains unchanged during the execution of one parallel region but may be different for another parallel region. The number of threads can be set with the clause

```
num_threads(expression)
```

The user-level library routine

```
int omp_get_num_threads()
```

returns the number of threads in the current team as integer value, which can be used in the code for SPMD computations. At the end of a parallel region there is an implicit barrier synchronization and the master thread is the only thread which continues the execution of the subsequent program code.

The clauses of a parallel directive include clauses which specify whether data will be private for each thread or shared among the threads executing the parallel region. Private variables of the threads of a parallel region are specified by the `private` clause with syntax

```
private(list_of_variables)
```

where `list_of_variables` is an arbitrary list of variables declared before. The `private` clause has the effect that for each private variable a new version of the original variable with the same type and size is created in the memory of each thread belonging to the parallel region. The private copy can be accessed and modified only by the thread owning the private copy. Shared variables of the team of threads are specified by the `shared` clause with the syntax

```
shared(list_of_variables)
```

where `list_of_variables` is a list of variables declared before. The effect of this clause is that the threads of the team access and modify the same original variable in the shared memory. The default clause can be used to specify whether variables in a parallel region are *shared* or *private* by default. The clause

```
default(shared)
```

causes all variables referenced in the construct to be shared except the private variables which are specified explicitly. The clause

```
default(none)
```

requires each variable in the construct to be specified explicitly as *shared* or *private*. The following example shows a first OpenMP program with a parallel region, in which multiple threads perform an SPMD computation on shared and private data.

Example The program code in Fig. 6.45 uses a `parallel` construct for a parallel SPMD execution on an array `x`. The input values are read in the function `initialize()` by the master thread. Within the parallel region the variables `x` and `npoints` are specified as *shared* and the variables `iam`, `np`, and `mypoints` are specified as *private*. All threads of the team of threads executing the parallel region store the number of threads in the variable `np` and their own thread id in the variable `iam`. The private variable `mypoints` is set to the number of points assigned to a thread. The function `compute_subdomain()` is executed by each thread of the team using its own private variables `iam` and `mypoints`. The actual computations are performed on the *shared* array `x`. □

```
#include <stdio.h>
#include <omp.h>

int npoints, iam, np, mypoints;
double *x;

int main() {
  scanf("%d", &npoints);
  x = (double *) malloc(npoints * sizeof(double));
  initialize();
  #pragma omp parallel shared(x,npoints) private(iam,np,mypoints)
  {
    np = omp_get_num_threads();
    iam = omp_get_thread_num();
    mypoints = npoints / np;
    compute_subdomain(x, iam, mypoints);
  }
}
```

Fig. 6.45 OpenMP program with `parallel` construct

A nesting of parallel regions by calling a `parallel` construct within a parallel region is possible. However, the default execution mode assigns only one thread to the team of the inner parallel region. The library function

```
void omp_set_nested(int nested)
```

with a parameter `nested` $\neq 0$ can be used to change the default execution mode to more than one thread for the inner region. The actual number of threads assigned to the inner region depends on the specific OpenMP implementation.

6.3.1.2 Parallel Loops

OpenMP provides constructs which can be used within a parallel region to distribute the work across threads that already exist in the team of threads executing the parallel region. The loop construct causes a distribution of the iterates of a **parallel loop** and has the syntax

```
#pragma omp for [clause [clause] ... ]
for (i = lower_bound; i op upper_bound; incr_expr) {
 { // loop iterate ... }
}
```

The use of the `for` construct is restricted to loops which are parallel loops, in which the iterates of the loop are independent of each other and for which the total number of iterates is known in advance. The effect of the `for` construct is that the iterates of the loop are assigned to the threads of the parallel region and are executed in parallel. The index variable `i` should not be changed within the loop and is considered as private variable of the thread executing the corresponding iterate. The expressions `lower_bound` and `upper_bound` are integer expressions, whose values should not be changed during the execution of the loop. The operator `op` is a boolean operator from the set { `<, <=, >, >=` }. The increment expression `incr_expr` can be of the form

```
++i, i++, --i, i--, i += incr, i -= incr,
i = i + incr, i = incr + i, i = i - incr,
```

with an integer expression `incr` that remains unchanged within the loop. The parallel loop of a `for` construct should not be finished with a `break` command. The parallel loop ends with an implicit synchronization of all threads executing the loop, and the program code following the parallel loop is only executed if all threads have finished the loop. The `nowait` clause given as clause of the `for` construct can be used to avoid this synchronization.

The specific distribution of iterates to threads is done by a scheduling strategy. OpenMP supports different scheduling strategies specified by the `schedule` parameters of the following list:

- `schedule(static, block_size)` specifies a static distribution of iterates to threads which assigns blocks of size `block_size` in a *round-robin* fashion to the threads available. When `block_size` is not given, blocks of almost equal size are formed and assigned to the threads in a blockwise distribution.

- `schedule(dynamic, block_size)` specifies a dynamic distribution of blocks to threads. A new block of size `block_size` is assigned to a thread as soon as the thread has finished the computation of the previously assigned block. When `block_size` is not provided, blocks of size one, i.e., consisting of only one iterate, are used.
- `schedule(guided, block_size)` specifies a dynamic scheduling of blocks with decreasing size. For the parameter value `block_size =1`, the new block assigned to a thread has a size which is the quotient of the number of iterates not assigned yet and the number of threads executing the parallel loop. For a parameter value `block_size = k > 1`, the size of the blocks is determined in the same way, but a block never contains fewer than k iterates (except for the last block which may contain fewer than `k` iterates). When no `block_size` is given, the blocks consist of one iterate each.
- `schedule(auto)` delegates the scheduling decision to the compiler and/or runtime system. Thus, any possible mapping of iterates to threads can be chosen.
- `schedule(runtime)` specifies a scheduling at runtime. At runtime the environmental variable `OMP_SCHEDULE`, which has to contain a character string describing one of the formats given above, is evaluated. Examples are

```
setenv OMP_SCHEDULE "dynamic, 4"
setenv OMP_SCHEDULE "guided"
```

  When the variable `OMP_SCHEDULE` is not specified, the scheduling used depends on the specific implementation of the OpenMP library.

A `for` construct without any `schedule` parameter is executed according to a default scheduling method also depending on the specific implementation of the OpenMP library. The use of the `for` construct is illustrated with the following example coding a matrix multiplication.

Example The code fragment in Fig. 6.46 shows a multiplication of a 100×100 matrix `MA` with a 100×100 matrix `MB` resulting in a matrix `MC` of the same dimension. The parallel region specifies `MA, MB, MC` as shared variables and the indices `row, col,i` as private. The two parallel loops use `static` scheduling with blocks of `row`. The first parallel loop initializes the result matrix `MC` with 0. The second parallel loop performs the matrix multiplication in a nested `for` loop. The `for` construct applies to the first `for` loop with iteration variable `row` and, thus, the iterates of the parallel loop are the nested loops of the iteration variables `col` and `i`. The static scheduling leads to a row-blockwise computation of the matrix `MC`. The first loop ends with an implicit synchronization. Since it is not clear that the first and second parallel loops have exactly the same assignment of iterates to threads, a `nowait` clause should be avoided to guarantee that the initialization is finished before the multiplication starts. □

The nesting of the `for` construct within the same `parallel` construct is not allowed. The nesting of parallel loops can be achieved by nesting `parallel` constructs so that each `parallel` construct contains exactly one `for` construct. This is illustrated by the following example.

```
#include <omp.h>

double MA[100][100], MB[100][100], MC[100][100];
int i, row, col, size = 100;

int main() {
  read_input(MA, MB);
  #pragma omp parallel shared(MA,MB,MC,size) private(row,col,i)
  {
    #pragma omp for schedule(static)
    for (row = 0; row < size; row++) {
      for (col = 0; col < size; col++)
        MC[row][col] = 0.0;
    }
    #pragma omp for schedule(static)
    for (row = 0; row < size; row++) {
      for (col = 0; col < size; col++)
        for (i = 0; i < size; i++)
          MC[row][col] += MA[row][i] * MB[i][col];
    }
  }
  write_output(MC);
}
```

Fig. 6.46 OpenMP program for a parallel matrix multiplication using a parallel region with two inner `for` constructs

Example The program code in Fig. 6.47 shows a modified version of the matrix multiplication in the last example. Again, the `for` construct applies to the for loop with the iteration index `row`. The iterates of this parallel loop start with another `parallel` construct which contains a second `for` construct applying to the loop with iteration index `col`. This leads to a parallel computation, in which each entry of `MC` can be computed by a different thread. There is no need for a synchronization between initialization and computation.

The OpenMP program in Fig. 6.47 implements the same parallelism as the Pthreads program for matrix multiplication in Fig. 6.1, see p. 262. A difference between the two programs is that the Pthreads program starts the threads explicitly. The thread creation in the OpenMP program is done implicitly by the OpenMP library which deals with the implementation of the nested loop and guarantees the correct execution. Another difference is that there is a limitation for the number of threads in the Pthreads program. The matrix size 8×8 in the Pthreads program from Fig. 6.1 leads to a correct program. A matrix size 100×100, however, would lead to the start of 10,000 threads, which is too large for most Pthreads implementations. There is no such limitation in the OpenMP program.

Fig. 6.47 OpenMP program for a parallel matrix multiplication with nested parallel loops

```
#include <omp.h>

double MA[100][100], MB[100][100], MC[100][100];
int i, row, col, size = 100;

int main() {
  read_input(MA, MB);
  #pragma omp parallel private(row,col,i)
  {
    #pragma omp for schedule(static)
    for (row = 0; row < size; row++) {
      #pragma omp parallel shared(MA, MB, MC, size)
      {
        #pragma omp for schedule(static)
        for (col = 0; col < size; col++) {
          MC[row][col] = 0.0;
          for (i = 0; i < size; i++)
            MC[row][col] += MA[row][i] * MB[i][col];
        }
      }
    }
  }
  write_output(MC);
}
```

6.3.1.3 Non-iterative Work-Sharing Constructs

The OpenMP library provides the `sections` construct to distribute non-iterative tasks to threads. Within the `sections` construct different code blocks are indicated by the `section` construct as tasks to be distributed. The syntax for the use of a `sections` construct is the following:

```
#pragma omp sections [clause [clause] ... ]
{
 [#pragma omp section]
  { // structured block ... }
 [#pragma omp section
  { // structured block ... }
  ⋮
 ]
}
```

The `section` constructs denote structured blocks which are independent of each other and can be executed in parallel by different threads. Each structured block starts with `#pragma omp section`, which can be omitted for the first block. The `sections` construct ends with an implicit synchronization unless a `nowait` clause is specified.

6.3.1.4 Single Execution

The `single` construct is used to specify that a specific structured block is executed by only one thread of the team, which is not necessarily the master thread. This can be useful for tasks like control messages during a parallel execution. The `single` construct has the syntax

```
#pragma omp single [Parameter [Parameter] ... ]
{ // structured block ... }
```

and can be used within a parallel region. The `single` construct also ends with an implicit synchronization unless a `nowait` clause is specified. The execution of a structured block within a parallel region by the master thread only is specified by

```
#pragma omp master
 { // structured block ... }
```

All other threads ignore the construct. There is no implicit synchronization of the master threads and the other threads of the team.

6.3.1.5 Syntactic Abbreviations

OpenMP provides abbreviated syntax for parallel regions containing only one `for` construct or only one `sections` construct. A parallel region with one `for` construct can be specified as

```
#pragma omp parallel for [clause [clause] ··· ]
 for (i = lower_bound; i op upper_bound; incr_expr) {
  { // loop body ... }
}
```

All clauses of the `parallel` construct or the `for` construct can be used. A parallel region with only one `sections` construct can be specified as

```
#pragma omp parallel sections [clause [clause] ··· ]
{
 [#pragma omp section]
  { // structured block ... }
 [#pragma omp section
  { // structured block ... }
  ⋮
 ]
}
```

6.3.2 Execution Environment Routines

The OpenMP library provides several execution environment routines that can be used to query and control the parallel execution environment. We present a few of them. The function

```
void omp_set_dynamic (int dynamic_threads)
```

can be used to set a dynamic adjustment of the number of threads by the runtime system and is called outside a parallel region. A parameter value `dynamic_threads` $\neq 0$ allows the dynamic adjustment of the number of threads for the subsequent parallel region. However, the number of threads within the same parallel region remains constant. The parameter value `dynamic_threads` $= 0$ disables the dynamic adjustment of the number of threads. The default case depends on the specific OpenMP implementation. The routine

```
int omp_get_dynamic (void)
```

returns information about the current status of the dynamic adjustment. The return value 0 denotes that no dynamic adjustment is set; a return value $\neq 0$ denotes that the dynamic adjustment is set. The number of threads can be set with the routine

```
void omp_set_num_threads (int num_threads)
```

which has to be called outside a parallel region and influences the number of threads in the subsequent parallel region (without a `num_threads` clause). The effect of this routine depends on the status of the dynamic adjustment. If the dynamic adjustment is set, the value of the parameter `num_threads` is the maximum number of threads to be used. If the dynamic adjustment is not set, the value of `num_threads` denotes the number of threads to be used in the subsequent parallel region. The routine

```
void omp_set_nested (int nested)
```

influences the number of threads in nested parallel regions. The parameter value `nested` $= 0$ means that the execution of the inner parallel region is executed by one thread sequentially. This is the default. A parameter value `nested` $\neq 0$ allows a nested parallel execution and the runtime system can use more than one thread for the inner parallel region. The actual behavior depends on the implementation. The routine

```
int omp_get_nested (void)
```

returns the current status of the nesting strategy for nested parallel regions.

6.3.3 Coordination and Synchronization of Threads

A parallel region is executed by multiple threads accessing the same shared data, so that there is need for synchronization in order to protect critical regions or avoid race condition, see also Chap. 3. OpenMP offers several constructs which can be used for synchronization and coordination of threads within a parallel region. The `critical` construct specifies a **critical region** which can be executed only by a single thread at a time. The syntax is

```
#pragma omp critical [(name)]
  structured block
```

An optional `name` can be used to identify a specific critical region. When a thread encounters a `critical` construct, it waits until no other thread executes a critical region of the same name `name` and then executes the code of the critical region. Unnamed critical regions are considered to be one critical region with the same unspecified name. The `barrier` construct with syntax

```
#pragma omp barrier
```

can be used to synchronize the threads at a certain point of execution. At such an explicit `barrier` construct all threads wait until all other threads of the team have reached the barrier and only then they continue the execution of the subsequent program code. The `atomic` construct can be used to specify that a single assignment statement is an **atomic operation**. The syntax is

```
#pragma omp atomic
  statement
```

and can contain statements of the form

```
x binop= E,
x++, ++x, x--, --x,
```

with an arbitrary variable `x`, a scalar expression `E` not containing `x`, and a binary operator `binop` $\in \{$`+`, `-`, `*`, `/`, `&`, `^`, `|`, `<<`, `>>`$\}$. The `atomic` construct ensures that the storage location `x` addressed in the statement belonging to the construct is updated atomically, which means that the load and store operations for `x` are atomic but not the evaluation of the expression `E`. No interruption is allowed between the load and store operations for variable `x`. However, the `atomic` construct does not enforce exclusive access to `x` with respect to a critical region specified by a `critical` construct. An advantage of the `atomic` construct over the `critical` construct is that parts of an array variable can also be specified as being atomically updated. The use of a `critical` construct would protect the entire array.

Example The following example shows an atomic update of a single array element `a[index[i]] += b`.

```
extern float a[], *p=a, b; int index[];
#pragma omp atomic
 a[index[i]] += b;
#pragma omp atomic
 p[i] -= 1.0;
```
□

A typical calculation which needs to be synchronized is a **global reduction** operation performed in parallel by the threads of a team. For this kind of calculation OpenMP provides the `reduction` clause, which can be used for `parallel`, `sections`, and `for` constructs. The syntax of the clause is

```
reduction (op: list)
```

where `op` $\in\{$`+`, `-`, `*`, `&`, `^`, `|`, `&&`, `||`$\}$ is a reduction operator to be applied and `list` is a list of reduction variables which have to be declared as shared. For each of the variables in `list`, a private copy is created for each thread of the team. The private copies are initialized to the neutral element of the operation `op` and can be updated by the owning thread. At the end of the region for which the `reduction` clause is specified, the local values of the reduction variables are combined according to the operator `op` and the result of the reduction is written into the original shared variable. The OpenMP compiler creates efficient code for executing the global reduction operation. No additional synchronization, such as the `critical` construct, has to be used to guarantee a correct result of the reduction. The following example illustrates the accumulation of values.

Example Figure 6.48 shows the accumulation of values in a `for` construct with the results written into the variables `a`, `y`, and `am`. Local reduction operations are performed by the threads of the team executing the `for` construct using private copies of `a`, `y`, and `am` for the local results. It is possible that a reduction operation is performed within a function, such as the function `sum` used for the accumulation onto `y`. At the end of the `for` loop, the values of the private copies of `a`, `y`, and `am` are accumulated according to + or `||`, respectively, and the final values are written into the original shared variables `a`, `y`, and `am`. □

```
#pragma omp parallel for reduction (+: a,y) reduction (||: am)
for (i=0; i<n; i++) {
  a += b[i];
  y = sum (y, c[i]);
  am = am || b[i] == c[i];
}
```

Fig. 6.48 Program fragment for the use of the `reduction` clause

The shared memory model of OpenMP might also require to coordinate the memory view of the threads. OpenMP provides the `flush` construct with the syntax

```
#pragma omp flush [(list)]
```

to produce a consistent view of the memory where `list` is a list of variables whose values should be made consistent. For pointers in the list `list` only the pointer value is updated. If no list is given, all variables are updated. An inconsistent view can occur since modern computers provide memory hierarchies. Updates are usually done in the faster memory parts, like registers or caches, which are not immediately visible to all threads. OpenMP has a specific relaxed-consistency shared memory in which updated values are written back later. But to make sure at a specific program point that a value written by one thread is actually read by another thread, the `flush` construct has to be used. It should be noted that no synchronization is provided if several threads execute the `flush` construct.

Example Figure 6.49 shows an example adopted from the OpenMP specification [130]. Two threads i ($i = 0, 1$) compute `work[i]` of array `work` which is written back to memory by the `flush` construct. The following update of array `sync[iam]` indicates that the computation of `work[iam]` is ready and written back to memory. The array `sync` is also written back by a second `flush` construct. In the `while` loop, a thread waits for the other thread to have updated its part of `sync`. The array `work` is then used in the function `combine()` only after both threads have updated their elements of `work`. □

```
#pragma omp parallel private (iam, neighbor) shared (work, sync)
{
  iam = omp_get_thread_num();
  sync[iam] = 0;
  #pragma omp barrier
  work[iam] = do_work();
  #pragma omp flush (work)
  sync[iam] = 1;
  #pragma omp flush (sync)
  neighbor = (iam != 0) ? (iam - 1) : (omp_get_num_threads() - 1);
  while (sync[neighbor] == 0) {
    #pragma omp flush (sync)
    { }
  }
  combine (work[iam], work[neighbor]);
}
```

Fig. 6.49 Program fragment for the use of the `flush` construct

Besides the explicit `flush` construct there is an implicit flush at several points of the program code, which are

- a `barrier` construct;
- entry to and exit from a `critical` region;
- at the end of a `parallel` region;
- at the end of a `for`, `sections`, or `single` construct without `nowait` clause;
- entry and exit of `lock` routines (which will be introduced below).

6.3.3.1 Locking Mechanism

The OpenMP runtime system also provides runtime library functions for a synchronization of threads with the **locking mechanism**. The locking mechanism has been described in Sect. 4.3 and in this chapter for Pthreads and Java threads. The specific locking mechanism of the OpenMP library provides two kinds of lock variables on which the locking runtime routines operate. *Simple locks* of type `omp_lock_t` can be locked only once. *Nestable locks* of type `omp_nest_lock_t` can be locked multiple times by the same thread. OpenMP lock variables should be accessed only by OpenMP locking routines. A lock variable is initialized by one of the following initialization routines:

```
void omp_init_lock (omp_lock_t *lock)
void omp_init_nest_lock (omp_nest_lock_t *lock)
```

for simple and nestable locks, respectively. A lock variable is removed with the routines

```
void omp_destroy_lock (omp_lock_t *lock)
void omp_destroy_nest_lock (omp_nest_lock_t *lock).
```

An initialized lock variable can be in the states *locked* or *unlocked*. At the beginning, the lock variable is in the state *unlocked*. A lock variable can be used for the synchronization of threads by locking and unlocking. To lock a lock variable the functions

```
void omp_set_lock (omp_lock_t *lock)
void omp_set_nest_lock (omp_nest_lock_t *lock)
```

are provided. If the lock variable is available, the thread calling the lock routine locks the variable. Otherwise, the calling thread blocks. A simple lock is available when no other thread has locked the variable before without unlocking it. A nestable lock variable is available when no other thread has locked the variable without unlocking it or when the calling thread has locked the variable, i.e., multiple locks for one nestable variable by the same thread are possible counted by an internal counter. When a thread uses a lock routine to lock a variable successfully, this thread

is said to *own* the lock variable. A thread owning a lock variable can unlock this variable with the routines

```
void omp_unset_lock (omp_lock_t *lock)
void omp_unset_nest_lock (omp_nest_lock_t *lock).
```

For a nestable lock, the routine `omp_unset_nest_lock ()` decrements the internal counter of the lock. If the counter has the value 0 afterwards, the lock variable is in the state *unlocked*. The locking of a lock variable without a possible blocking of the calling thread can be performed by one of the routines

```
void omp_test_lock (omp_lock_t *lock)
void omp_test_nest_lock (omp_nest_lock_t *lock)
```

for simple and nestable lock variables, respectively. When the lock is available, the routines lock the variable or increment the internal counter and return a result value $\neq 1$. When the lock is not available, the `test` routine returns 0 and the calling thread is not blocked.

Example Figure 6.50 illustrates the use of nestable lock variables, see [130]. A data structure `pair` consists of two integers `a` and `b` and a nestable lock variable `l`, which is used to synchronize the updates of `a, b`, or the entire `pair`. It is assumed that the lock variable `l` has been initialized before calling `f()`. The increment functions `incr_a()` for incrementing `a`, `incr_b()` for incrementing `b`, and `incr_pair()` for incrementing both integer variables are given. The function `incr_a()` is only called from `incr_pair()` and does not need an additional locking. The functions `incr_b()` and `incr_pair()` are protected by the lock since they can be called concurrently. □

6.4 Exercises for Chap. 6

Exercise 6.1 Modify the matrix multiplication program from Fig. 6.1 on p. 262 so that a fixed number of threads is used for the multiplication of matrices of arbitrary size. For the modification, let each thread compute the rows of the result matrix instead of a single entry. Compute the number of rows that each thread must compute such that each thread has about the same number of rows to compute. Is there any synchronization required in the program?

Exercise 6.2 Use the task pool implementation from Sect. 6.1.6 on p. 276 to implement a parallel matrix multiplication. To do so, use the function `thread_mult()` from Fig. 6.1 to define a task as the computation of one entry of the result matrix and modify the function if necessary so that it fits to the requirements of the task pool. Modify the main program so that all tasks are generated and inserted into the task pool before the threads to perform the computations are started. Measure the

Fig. 6.50 Program fragment illustrating the use of nestable lock variables

```
#include <omp.h>

typedef struct {
  int a, b;
  omp_nest_lock_t l;
} pair;

void incr_a (pair *p, int a) {
  p->a += a;
}

void incr_b (pair *p, int b) {
  omp_set_nest_lock (&p->l);
  p->b += b;
  omp_unset_nest_lock (&p->l);
}

void incr_pair (pair *p, int a, int b) {
  omp_set_nest_lock (&p->l);
  incr_a (p, a);
  incr_b (p, b);
  omp_unset_nest_lock (&p->l);
}

void f (pair *p) {
  extern int work1(), work2(), work3();
  #pragma omp parallel sections
  {
    #pragma omp section
      incr_pair (p, work1(), work2());
    #pragma omp section
      incr_b (p,work3());
  }
}
```

resulting execution time for different numbers of threads and different matrix sizes and compare the execution time with the execution time of the implementation of the last exercise.

Exercise 6.3 Consider the r/w lock mechanism in Fig. 6.5. The implementation given does not provide operations that are equivalent to the function `pthread_mutex_trylock()`. Extend the implementation from Fig. 6.5 by specifying functions `rw_lock_rtrylock()` and `rw_lock_wtrylock()` which return `EBUSY` if the requested read or write permit cannot be granted.

Exercise 6.4 Consider the r/w lock mechanism in Fig. 6.5. The implementation given favors read requests over write requests in the sense that a thread will get a write permit only if no other thread requests a read permit, but read permits are given without waiting also in the presence of other read permits. Change the implementation such that write permits have priority, i.e., as soon as a write permit

arrives, no more read permits are granted until the write permit has been granted and the corresponding write operation is finished. To test the new implementation write a program which starts three threads, two read threads, and one write thread. The first read thread requests five read permits one after another. As soon as it gets the read permits it prints a control message and waits for 2 s (use `sleep(2)`) before requesting the next read permit. The second read thread does the same except that it only waits 1 s after the first read permit and 2 s otherwise. The write thread first waits 5 s and then requests a write permit and prints a control message after it has obtained the write permit; then the write permit is released again immediately.

Exercise 6.5 An r/w lock mechanism allows multiple readers to access a data structure concurrently, but only a single writer is allowed to access the data structures at a time. We have seen a simple implementation of r/w locks in Pthreads in Fig. 6.5. Transfer this implementation to Java threads by writing a new class `RWlock` with entries `num_r` and `num_w` to count the current number of read and write permits given. The class `RWlock` should provide methods similar to the functions in Fig. 6.5 to request or release a read or write permit.

Exercise 6.6 Consider the pipelining programming pattern and its Pthreads implementation in Sect. 6.1.7. In the example given, each pipeline stage adds 1 to the integer value received from the predecessor stage. Modify the example such that pipeline stage i adds the value i to the value received from the predecessor. In the modification, there should still be only one function `pipe_stage()` expressing the computations of a pipeline stage. This function must receive an appropriate parameter for the modification.

Exercise 6.7 Use the task pool implementation from Sect. 6.1.6 to define a parallel loop pattern. The loop body should be specified as function with the loop variable as parameter. The iteration space of the parallel loop is defined as the set of all values that the loop variable can have. To execute a parallel loop, all possible indices are stored in a parallel data structure similar to a task pool which can be accessed by all threads. For the access, a suitable synchronization must be used.

(a) Modify the task pool implementation accordingly such that functions for the definition of a parallel loop and for retrieving an iteration from the parallel loop are provided. The thread function should also be provided.
(b) The parallel loop pattern from (a) performs a dynamic load balancing since a thread can retrieve the next iteration as soon as its current iteration is finished. Modify this operation such that a thread retrieves a chunk of iterations instead of a single operation to reduce the overhead of load balancing for fine-grained iterations.
(c) Include guided self-scheduling (GSS) in your parallel loop pattern. GSS adapts the number of iterations retrieved by a thread to the total number of iterations that are still available. If n threads are used and there are R_i remaining iterations, the next thread retrieves

$$x_i = \left\lceil \frac{R_i}{n} \right\rceil$$

iterations, $i = 1, 2, \ldots$. For the next retrieval, $R_{i+1} = R_i - x_i$ iterations remain. R_1 is the initial number of iterations to be executed.

(d) Use the parallel loop pattern to express the computation of a matrix multiplication where the computation of each matrix entry can be expressed as an iteration of a parallel loop. Measure the resulting execution time for different matrix sizes. Compare the execution time for the two load balancing schemes (standard and GSS) implemented.

Exercise 6.8 Consider the client–server pattern and its Pthreads implementation in Sect. 6.1.8. Extend the implementation given in this section by allowing a cancellation with deferred characteristics. To be cancellation-safe, mutex variables that have been locked must be released again by an appropriate cleanup handler. When a cancellation occurs, allocated memory space should also be released. In the server function `tty_server_routine()`, the variable `running` should be reset when a cancellation occurs. Note that this may create a concurrent access. If a cancellation request arrives during the execution of a synchronous request of a client, the client thread should be informed that a cancellation has occurred. For a cancellation in the function `client_routine()`, the counter `client_threads` should be kept consistent.

Exercise 6.9 Consider the task pool pattern and its implementation in Pthreads in Sect. 6.1.6. Implement a Java class `TaskPool` with the same functionality. The task pool should accept each object of a class which implements the interface `Runnable` as task. The tasks should be stored in an array `final Runnable tasks[]`. A constructor `TaskPool(int p, int n)` should be implemented that allocates a task array of size `n` and creates `p` threads which access the task pool. The methods `run()` and `insert(Runnable w)` should be implemented according to the Pthreads functions `tpool_thread()` and `tpool_insert()` from Fig. 6.7. Additionally, a method `terminate()` should be provided to terminate the threads that have been started in the constructor. For each access to the task pool, a thread should check whether a termination request has been set.

Exercise 6.10 Transfer the pipelining pattern from Sect. 6.1.7 for which Figs. 6.8, 6.9, 6.10, and 6.11 give an implementation in Pthreads to Java. For the Java implementation, define classes for a pipeline stage as well as for the entire pipeline which provide the appropriate method to perform the computation of a pipeline stage, to send data into the pipeline, and to retrieve a result from the last stage of the pipeline.

Exercise 6.11 Transfer the client–server pattern for which Figs. 6.13, 6.14, 6.15, and 6.16 give a Pthreads implementation to Java threads. Define classes to store a request and for the server implementation explain the synchronizations performed and give reasons that no deadlock can occur.

Exercise 6.12 Consider the following OpenMP program piece:

```
int x=0;
int y=0;

void foo1() {
#pragma omp critical (x)
     { foo2(); x+=1; }
}
void foo2() {
#pragma omp critical(y)
     { y+=1; }
}
void foo3() {
#pragma omp critical(y)
     { y-=1; foo4(); }
}
void foo4() {
#pragma omp critical(x)
     { x-=1; }
}
int main(int argx, char **argv) {
     int x;
#pragma omp parallel private(i) {
     for (i=0; i<10; i++)
         { foo1(), foo3(); }
}
printf(''%d %d \n'', x,y )
}
```

We assume that two threads execute this piece of code on two cores of a multicore processor. Can a deadlock situation occur? If so, describe the execution order which leads to the deadlock. If not, give reasons why a deadlock is not possible.

Chapter 7
Algorithms for Systems of Linear Equations

The solution of a system of simultaneous linear equations is a fundamental problem in numerical linear algebra and is a basic ingredient of many scientific simulations. Examples are scientific or engineering problems modeled by ordinary or partial differential equations. The numerical solution is often based on discretization methods leading to a system of linear equations. In this chapter, we present several standard methods for solving systems of linear equations of the form

$$Ax = b, \tag{7.1}$$

where $A \in \mathbb{R}^{n \times n}$ is an $(n \times n)$ matrix of real numbers, $b \in \mathbb{R}^n$ is a vector of size n, and $x \in \mathbb{R}^n$ is an unknown solution vector of size n specified by the linear system (7.1) to be determined by a solution method. There exists a solution x for Eq. (7.1) if the matrix A is non-singular, which means that a matrix A^{-1} with $A \cdot A^{-1} = I$ exists; I denotes the n-dimensional identity matrix and $\cdot$ denotes the matrix product. Equivalently, the determinant of matrix A is not equal to zero. For the exact mathematical properties we refer to a standard book for linear algebra [71]. The emphasis of the presentation in this chapter is on parallel implementation schemes for linear system solvers.

The solution methods for linear systems are classified as direct and iterative. **Direct solution methods** determine the exact solution (except rounding errors) in a fixed number of steps depending on the size n of the system. Elimination methods and factorization methods are considered in the following. **Iterative solution methods** determine an approximation of the exact solution. Starting with a start vector, a sequence of vectors is computed which converges to the exact solution. The computation is stopped if the approximation has an acceptable precision. Often, iterative solution methods are faster than direct methods and their parallel implementation is straightforward. On the other hand, the system of linear equations needs to fulfill some mathematical properties in order to guarantee the convergence to the exact solution. For sparse matrices, in which many entries are zeros, there is an advantage for iterative methods since they avoid a fill-in of the matrix with non-zero elements.

This chapter starts with a presentation of Gaussian elimination, a direct solver, and its parallel implementation with different data distribution patterns. In Sect. 7.2, direct solution methods for linear systems with tridiagonal structure or banded

T. Rauber, G. Rünger, *Parallel Programming*,
DOI 10.1007/978-3-642-04818-0_7, © Springer-Verlag Berlin Heidelberg 2010

matrices, in particular cyclic reduction and recursive doubling, are discussed. Section 7.3 is devoted to iterative solution methods, Sect. 7.5 presents the Cholesky factorization, and Sect. 7.4 introduces the conjugate gradient method. All presentations mainly concentrate on the aspects of a parallel implementation.

7.1 Gaussian Elimination

For the well-known Gaussian elimination, we briefly present the sequential method and then discuss parallel implementations with different data distributions. The section closes with an analysis of the parallel runtime of the Gaussian elimination with double-cyclic data distribution.

7.1.1 Gaussian Elimination and LU Decomposition

Written out in full the linear system $Ax = b$ has the form

$$\begin{array}{cccccc} a_{11}x_1 & + a_{12}x_2 & + \cdots + & a_{1n}x_n & = & b_1 \\ \vdots & \vdots & & \vdots & \vdots & \\ a_{i1}x_1 & + a_{i2}x_2 & + \cdots + & a_{in}x_n & = & b_i \\ \vdots & \vdots & & \vdots & \vdots & \\ a_{n1}x_1 & + a_{n2}x_2 & + \cdots + & a_{nn}x_n & = & b_n \end{array} .$$

The Gaussian elimination consists of two phases, the forward elimination and the backward substitution. The forward elimination transforms the linear system (7.1) into a linear system $Ux = b'$ with an $(n \times n)$ matrix U in upper triangular form. The transformation employs reordering of equations or operations that add a multiple of one equation to another. Hence, the solution vector remains unchanged. In detail, the **forward elimination** performs the following $n - 1$ steps. The matrix $A^{(1)} := A = (a_{ij})$ and the vector $b^{(1)} := b = (b_i)$ are subsequently transformed into matrices $A^{(2)}, \ldots, A^{(n)}$ and vectors $b^{(2)}, \ldots, b^{(n)}$, respectively. The linear equation systems $A^{(k)}x = b^{(k)}$ have the same solution as Eq. (7.1) for $k = 2, \ldots, n$. The matrix $A^{(k)}$ computed in step $k - 1$ has the form

$$A^{(k)} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1,k-1} & a_{1k} & \cdots & a_{1n} \\ 0 & a_{22}^{(2)} & \cdots & a_{2,k-1}^{(2)} & a_{2k}^{(2)} & \cdots & a_{2n}^{(2)} \\ \vdots & \ddots & \ddots & \vdots & \vdots & & \vdots \\ \vdots & & \ddots & a_{k-1,k-1}^{(k-1)} & a_{k-1,k}^{(k-1)} & \cdots & a_{k-1,n}^{(k-1)} \\ \vdots & & & 0 & a_{kk}^{(k)} & \cdots & a_{kn}^{(k)} \\ \vdots & & & \vdots & \vdots & \ddots & \vdots \\ 0 & \cdots & \cdots & 0 & a_{nk}^{(k)} & \cdots & a_{nn}^{(k)} \end{bmatrix} .$$

The first $k-1$ rows are identical to the rows in matrix $A^{(k-1)}$. In the first $k-1$ columns, all elements below the diagonal element are zero. Thus, the last matrix $A^{(n)}$ has upper triangular form. The matrix $A^{(k+1)}$ and the vector $b^{(k+1)}$ are calculated from $A^{(k)}$ and $b^{(k)}$, $k = 1, \ldots, n-1$, by subtracting suitable multiples of row k of $A^{(k)}$ and element k of $b^{(k)}$ from the rows $k+1, k+2, \ldots, n$ of A and elements $b_{k+1}^{(k)}, b_{k+2}^{(k)}, \ldots, b_n^{(k)}$, respectively. The elimination factors for row i are

$$l_{ik} = a_{ik}^{(k)} / a_{kk}^{(k)}, \qquad i = k+1, \ldots, n \; . \tag{7.2}$$

They are chosen such that the coefficient of x_k of the unknown vector x is eliminated from equations $k+1, k+2, \ldots, n$. The rows of $A^{(k+1)}$ and the entries of $b^{(k+1)}$ are calculated according to

$$a_{ij}^{(k+1)} = a_{ij}^{(k)} - l_{ik} a_{kj}^{(k)}, \tag{7.3}$$

$$b_i^{(k+1)} = b_i^{(k)} - l_{ik} b_k^{(k)} \tag{7.4}$$

for $k < j \leq n$ and $k < i \leq n$. Using the equation system $A^{(n)} x = b^{(n)}$, the result vector x is calculated in the **backward substitution** in the order $x_n, x_{n-1}, \ldots, x_1$ according to

$$x_k = \frac{1}{a_{kk}^{(n)}} \left(b_k^{(n)} - \sum_{j=k+1}^{n} a_{kj}^{(n)} x_j \right). \tag{7.5}$$

Figure 7.1 shows a program fragment in C for a sequential Gaussian elimination. The inner loop computing the matrix elements is iterated approximately k^2 times so that the entire loop has runtime $\sum_{k=1}^{n} k^2 = \frac{1}{6} n(n+1)(2n+1) \approx n^3/3$ which leads to an asymptotic runtime $O(n^3)$.

7.1.1.1 LU Decomposition and Triangularization

The matrix A can be represented as the matrix product of an upper triangular matrix $U := A^{(n)}$ and a lower triangular matrix L which consists of the elimination factors (7.2) in the following way:

$$L = \begin{bmatrix} 1 & 0 & 0 & \ldots & & 0 \\ l_{21} & 1 & 0 & \ldots & & 0 \\ l_{31} & l_{32} & 1 & & & 0 \\ \vdots & \vdots & \ddots & & \ddots & 0 \\ l_{n1} & l_{n2} & l_{n3} & \ldots & l_{n,n-1} & 1 \end{bmatrix}.$$

The matrix representation $A = L \cdot U$ is called triangularization or LU decomposition. When only the LU decomposition is needed, the right-hand side of the linear

```
double *gauss_sequential (double **a, double *b)
{
  double *x, sum, l[MAX_SIZE];
  int i,j,k,r;

  x = (double *) malloc(n * sizeof(double));
  for (k = 0; k < n-1; k++) { /* Forward elimination */
     r = max_col(a,k);
     if (k != r) exchange_row(a,b,r,k);
     for (i=k+1; i < n; i++) {
        l[i] = a[i][k]/a[k][k];
        for (j=k+1; j < n; j++)
           a[i][j] = a[i][j] - l[i] * a[k][j];
        b[i] = b[i] - l[i] * b[k];
     }
  }
  for (k = n-1; k >= 0; k--) { /* Backward substitution */
     sum = 0.0;
     for (j=k+1; j < n; j++)
        sum = sum + a[k][j] * x[j];
     x[k] = 1/a[k][k] * (b[k] - sum);
  }
  return x;
}
```

Fig. 7.1 Program fragment in C notation for a sequential Gaussian elimination of the linear system $Ax = b$. The matrix A is stored in array `a`, the vector b is stored in array `b`. The indices start with 0. The functions `max_col(a,k)` and `exchange_row(a,b,r,k)` implement pivoting. The function `max_col(a,k)` returns the index r with $|a_{rk}| = \max_{k \le s \le n}(|a_{sk}|)$. The function `exchange_row(a,b,r,k)` exchanges the rows r and k of A and the corresponding elements b_r and b_k of the right-hand side

system does not have to be transformed. Using the LU decomposition, the linear system $Ax = b$ can be rewritten as

$$Ax = LA^{(n)}x = Ly = b \quad \text{with} \quad y = A^{(n)}x \tag{7.6}$$

and the solution can be determined in two steps. In the first step, the vector y is obtained by solving the triangular system $Ly = b$ by forward substitution. The forward substitution corresponds to the calculation of $y = b^{(n)}$ from Eq. (7.4). In the second step, the vector x is determined from the upper triangular system $A^{(n)}x = y$ by backward substitution. The advantage of the LU factorization over the elimination method is that the factorization into L and U is done only once but

can be used to solve several linear systems with the same matrix A and different right-hand side vectors b without repeating the elimination process.

7.1.1.2 Pivoting

Forward elimination and LU decomposition require the division by $a_{kk}^{(k)}$ and so these methods can only be applied when $a_{kk}^{(k)} \neq 0$. That is, even if $\det A \neq 0$ and the system $Ax = y$ is solvable, there does not need to exist a decomposition $A = LU$ when $a_{kk}^{(k)}$ is a zero element. However, for a solvable linear system, there exists a matrix resulting from permutations of rows of A, for which an LU decomposition is possible, i.e., $BA = LU$ with a permutation matrix B describing the permutation of rows of A. The permutation of rows of A, if necessary, is included in the elimination process. In each elimination step, a **pivot element** is determined to substitute $a_{kk}^{(k)}$. A pivot element is needed when $a_{kk}^{(k)} = 0$ and when $a_{kk}^{(k)}$ is very small which would induce an elimination factor, which is very large leading to imprecise computations. Pivoting strategies are used to find an appropriate pivot element. Typical strategies are column pivoting, row pivoting, and total pivoting.

Column pivoting considers the elements $a_{kk}^{(k)} \cdots a_{nk}^{(k)}$ of column k and determines the element $a_{rk}^{(k)}$, $k \leq r \leq n$, with the maximum absolute value. If $r \neq k$, the rows r and k of matrix $A^{(k)}$ and the values $b_k^{(k)}$ and $b_r^{(k)}$ of the vector $b^{(k)}$ are exchanged. **Row pivoting** determines a pivot element $a_{kr}^{(k)}$, $k \leq r \leq n$, within the elements $a_{kk}^{(k)} \cdots a_{kn}^{(k)}$ of row k of matrix $A^{(k)}$ with the maximum absolute value. If $r \neq k$, the columns k and r of $A^{(k)}$ are exchanged. This corresponds to an exchange of the enumeration of the unknowns x_k and x_r of vector x. **Total pivoting** determines the element with the maximum absolute value in the matrix $\tilde{A}^{(k)} = (a_{ij}^{(k)})$, $k \leq i, j \leq n$, and exchanges columns and rows of $A^{(k)}$ depending on $i \neq k$ and $j \neq k$. In practice, row or column pivoting is used instead of total pivoting, since they have smaller computation time, and total pivoting may also destroy special matrix structures like banded structures.

The implementation of pivoting avoids the actual exchange of rows or columns in memory and uses index vectors pointing to the current rows of the matrix. The indexed access to matrix elements is more expensive but in total the indexed access is usually less expensive than moving entire rows in each elimination step. When supported by the programming language, a dynamic data storage in the form of separate vectors for rows of the matrix, which can be accessed through a vector pointing to the rows, may lead to more efficient implementations. The advantage is that matrix elements can still be accessed with a two-dimensional index expression but the exchange of rows corresponds to a simple exchange of pointers.

7.1.2 Parallel Row-Cyclic Implementation

A parallel implementation of the Gaussian elimination is based on a data distribution of matrix A and of the sequence of matrices $A^{(k)}$, $k = 2, \ldots, n$, which can be a row-oriented, a column-oriented, or a checkerboard distribution, see Sect. 3.4. In this section, we consider a row-oriented distribution.

From the structure of the matrices $A^{(k)}$ it can be seen that a blockwise row-oriented data distribution is not suitable because of load imbalances: For a blockwise row-oriented distribution processor P_q, $1 \leq q \leq p$, owns the rows $(q-1) \cdot n/p + 1, \ldots, q \cdot n/p$ so that after the computation of $A^{(k)}$ with $k = q \cdot n/p + 1$ there is no computation left for this processor and it becomes idle. For a row-cyclic distribution, there is a better load balance, since processor P_q, $1 \leq q \leq p$, owns the rows $q, q+p, q+2p, \ldots$, i.e., it owns all rows i with $1 \leq i \leq n$, and $q = ((i-1) \bmod p)+1$. The processors begin to get idle only after the first $n-p$ stages, which is reasonable for $p \ll n$. Thus, we consider a parallel implementation of the Gaussian elimination with a row-cyclic distribution of matrix **A** and a column-oriented pivoting. One step of the forward elimination computing $A^{(k+1)}$ and $b^{(k+1)}$ for given $A^{(k)}$ and $b^{(k)}$ performs the following computation and communication phases:

1. **Determination of the local pivot element:** Each processor considers its local elements of column k in the rows $k, \ldots, n$ and determines the element (and its position) with the largest absolute value.
2. **Determination of the global pivot element:** The global pivot element is the local pivot element which has the largest absolute value. A single-accumulation operation with the maximum operation as reduction determines this global pivot element. The root processor of this global communication operation sends the result to all other processors.
3. **Exchange of the pivot row:** If $k \neq r$ for a pivot element $a_{rk}^{(k)}$, the row k owned by processor P_q and the pivot row r owned by processor $P_{q'}$ have to be exchanged. When $q = q'$, the exchange can be done locally by processor P_q. When $q \neq q'$, then communication with single transfer operations is required. The elements b_k and b_r are exchanged accordingly.
4. **Distribution of the pivot row:** Since the pivot row (now row k) is required by all processors for the local elimination operations, processor P_q sends the elements $a_{kk}^{(k)}, \ldots, a_{kn}^{(k)}$ of row k and the element $b_k^{(k)}$ to all other processors.
5. **Computation of the elimination factors:** Each processor locally computes the elimination factors l_{ik} for which it owns the row i according to Formula (7.2).
6. **Computation of the matrix elements:** Each processor locally computes the elements of $A^{(k+1)}$ and $b^{(k+1)}$ using its elements of $A^{(k)}$ and $b^{(k)}$ according to Formulas (7.3) and (7.4).

The computation of the solution vector x in the backward substitution is inherently sequential, since the values x_k, $k = n, \ldots, 1$, depend on each other and are computed one after another. In step k, processor P_q owning row k computes the value x_k according to Formula (7.5) and sends the value to all other processors by a single-broadcast operation.

A program fragment implementing the computation phases 1–6 and the backward substitution is given in Fig. 7.2. The matrix A and the vector b are stored in a two- and a one-dimensional array `a` and `b`, respectively. Some of the local functions are already introduced in the program in Fig. 7.1. The SPMD program uses the variable `me` to store the individual processor number. This processor number, the

```
double *gauss_cyclic (double **a, double *b)
{
  double *x, l[MAX_SIZE], *buf;
  int i,j,k,r, tag=42;
  MPI_Status status;
  struct { double val; int node; } z,y;
  x = (double *) malloc(n * sizeof(double));
  buf = (double *) malloc((n+1) * sizeof(double));
  for (k=0 ; k<n-1 ; k++) { /* Forward elimination */
    r = max_col_loc(a,k);
    z.node = me;
    if (r != -1) z.val = fabs(a[r][k]); else z.val = 0.0;
    MPI_Allreduce(&z,&y,1,MPI_DOUBLE_INT,MPI_MAXLOC,MPI_COMM_WORLD);
    if (k % p == y.node){ /* Pivot row and row k are on the same processor */
       if (k % p == me) {
          if (a[k][k] != y.val) exchange_row(a,b,r,k);
          copy_row(a,b,k,buf);
       }
    }
    else /* Pivot row and row k are owned by different processors */
      if (k % p == me) {
        copy_row(a,b,k,buf);
        MPI_Send(buf+k,n-k+1,MPI_DOUBLE,y.node,tag,
                              MPI_COMM_WORLD);
      }
      else if (y.node == me) {
        MPI_Recv(buf+k,n-k+1,MPI_DOUBLE,MPI_ANY_SOURCE,
                              tag,MPI_COMM_WORLD,&status);
        copy_exchange_row(a,b,r,buf,k);
      }
    MPI_Bcast(buf+k,n-k+1,MPI_DOUBLE,y.node,MPI_COMM_WORLD);
    if ((k % p != y.node) && (k % p == me)) copy_back_row(a,b,buf,k);
    i = k+1; while (i % p != me) i++;
    for ( ; i<n; i+=p) {
      l[i] = a[i][k] / buf[k];
      for (j=k+1; j<n; j++)
        a[i][j] = a[i][j] - l[i]*buf[j];
      b[i] = b[i] - l[i]*buf[n];
    }
  }
  for (k=n-1; k>=0; k--) { /* Backward substitution */
    if (k % p == me) {
      sum = 0.0;
      for (j=k+1; j < n; j++)  sum = sum + a[k][j] * x[j];
      x[k] = 1/a[k][k] * (b[k] - sum); }
    MPI_Bcast(&x[k],1,MPI_DOUBLE,k%p,MPI_COMM_WORLD);
  }
  return x;
}
```

Fig. 7.2 Program fragment with C notation and MPI operations for the Gaussian elimination with row-cyclic distribution

current value of k, and the pivot row are used to distinguish between different computations for the single processors. The global variables `n` and `p` are the system size and the number of processors executing the parallel program. The parallel algorithm is implemented in the program in Fig. 7.2 in the following way:

1. **Determination of the local pivot element:** The function `max_col_loc(a,k)` determines the row index r of the element `a[r][k]`, which has the largest local absolute value in column k for the rows $\geq k$. When a processor has no element of column k for rows $\geq k$, the function returns -1.
2. **Determination of the global pivot element:** The global pivoting is performed by an `MPI_Allreduce()` operation, implementing a single-accumulation with a subsequent single-broadcast. The MPI reduction operation `MPI_MAXLOC` for data type `MPI_DOUBLE_INT` consisting of one double value and one integer value is used. The MPI operations have been introduced in Sect. 5.2. The `MPI_Allreduce()` operation returns `y` with the pivot element in `y.val` and the processor owning the corresponding row in `y.node`. Thus, after this step all processors know the global pivot element and the owner for possible communication.
3. **Exchange of the pivot row:** Two cases are considered:
   - If the owner of the pivot row is the processor also owning row k (i.e., `k%p == y.node`), the rows k and r are exchanged locally by this processor for $r \neq k$. Row k is now the pivot row. The function `copy_row(a,b,k,buf)` copies the pivot row into the buffer `buf`, which is used for further communication.
   - If different processors own the row k and the pivot row r, row k is sent to the processor `y.node` owning the pivot row with `MPI_Send` and `MPI_Recv` operations. Before the send operation, the function `copy_row(a,b,k,buf)` copies row k of array `a` and element k of array `b` into a common buffer `buf` so that only one communication operation needs to be applied. After the communication, the processor `y.node` finalizes its exchange with the pivot row. The function `copy_exchange_row(a,b,r,buf,k)` exchanges the row r (still the pivot row) and the buffer `buf`. The appropriate row index r is known from the former local determination of the pivot row. Now the former row k is the row r and the buffer `buf` contains the pivot row.

   Thus, in both cases the pivot row is stored in buffer `buf`.
4. **Distribution of the pivot row:** Processor `y.node` sends the buffer `buf` to all other processors by an `MPI_Bcast()` operation. For the case of the pivot row being owned by a different processor than the owner of row k, the content of `buf` is copied into row k by this processor using `copy_back_row()`.
5. and 6. **Computation of the elimination factors and the matrix elements:** The computation of the elimination factors and the new arrays `a` and `b` is done in parallel. Processor P_q starts this computation with the first row $i > k$ with i mod $p = q$.

For a row-cyclic implementation of the Gaussian elimination, an alternative way of storing array `a` and vector `b` can be used. The alternative data structure consists

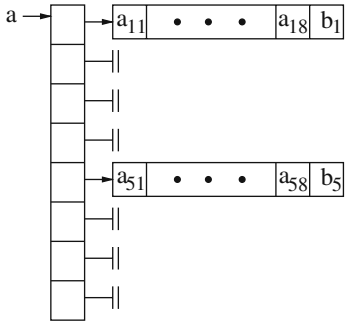


Fig. 7.3 Data structure for the Gaussian elimination with $n = 8$ and $p = 4$ showing the rows stored by processor P_1. Each row stores $n + 1$ elements consisting of one row of the matrix `a` and the corresponding element of `b`

of a one-dimensional array of pointers and n one-dimensional arrays of length $n+1$ each containing one row of `a` and the corresponding element of `b`. The entries in the pointer-array point to the row-arrays. This storage scheme not only facilitates the exchange of rows but is also convenient for a distributed storage. For a distributed memory, each processor P_q stores the entire array of pointers but only the rows i with i mod $p = q$; all other pointers are `NULL`-pointers. Figure 7.3 illustrates this storage scheme for $n = 8$. The advantage of storing an element of `b` together with `a` is that the copy operation into a common buffer can be avoided. Also the computation of the new values for `a` and `b` is now only one loop with $n+1$ iterations. This implementation variant is not shown in Fig. 7.2.

7.1.3 Parallel Implementation with Checkerboard Distribution

A parallel implementation using a block–cyclic checkerboard distribution for matrix A can be described with the parameterized data distribution introduced in Sect. 3.4. The parameterized data distribution is given by a distribution vector

$$((p_1, b_1), (p_2, b_2)) \tag{7.7}$$

with a $p_1 \times p_2$ virtual processor mesh with p_1 rows, p_2 columns, and $p_1 \cdot p_2 = p$ processors. The numbers b_1 and b_2 are the sizes of a block of data with b_1 rows and b_2 columns. The function $\mathcal{G} : P \to \mathbb{N}^2$ maps each processor to a unique position in the processor mesh. This leads to the definition of p_1 **row groups**

$$R_q = \{Q \in P \mid \mathcal{G}(Q) = (q, \cdot)\}$$

with $|R_q| = p_2$ for $1 \leq q \leq p_1$ and p_2 **column groups**

$$C_q = \{Q \in P \mid \mathcal{G}(Q) = (\cdot, q)\}$$

with $|C_q| = p_1$ for $1 \leq q \leq p_2$. The row groups as well as the column groups are a partition of the entire set of processors, i.e.,

$$\bigcup_{q=1}^{p_1} R_q = \bigcup_{q=1}^{p_2} C_q = P$$

and $R_q \cap R_{q'} = \emptyset = C_q \cap C_{q'}$ for $q \neq q'$. Row i of the matrix A is distributed across the local memories of the processors of only one row group, denoted $Ro(i)$ in the following. This is the row group R_k with $k = \left\lfloor \frac{i-1}{b_1} \right\rfloor \bmod p_1 + 1$. Analogously, column j is distributed within one column group, denoted as $Co(j)$, which is the column group C_k with $k = \left\lfloor \frac{j-1}{b_2} \right\rfloor \bmod p_2 + 1$.

Example For a matrix of size 12×12 (i.e., $n = 12$), $p = 4$ processors $\{P_1, P_2, P_3, P_4\}$ and distribution vector $((p_1, b_1), (p_2, b_2)) = ((2, 2), (2, 3))$, the virtual processor mesh has size 2×2 and the data blocks have size 2×3. There are two row groups and two column groups:

$$\begin{aligned} R_1 &= \{Q \in P \mid \mathcal{G}(Q) = (1, j), j = 1, 2\}, \\ R_2 &= \{Q \in P \mid \mathcal{G}(Q) = (2, j), j = 1, 2\}, \\ C_1 &= \{Q \in P \mid \mathcal{G}(Q) = (j, 1), j = 1, 2\}, \\ C_2 &= \{Q \in P \mid \mathcal{G}(Q) = (j, 2), j = 1, 2\} . \end{aligned}$$

The distribution of matrix A is shown in Fig. 7.4. It can be seen that row 5 is distributed in row group R_1 and that column 7 is distributed in column group C_1.□

Using a checkerboard distribution with distribution vector (7.7), the computation of $A^{(k)}$ has the following implementation, which has a different communication pattern than the previous implementation. Figure 7.5 illustrates the communication and computation phases of the Gaussian elimination with checkerboard distribution.

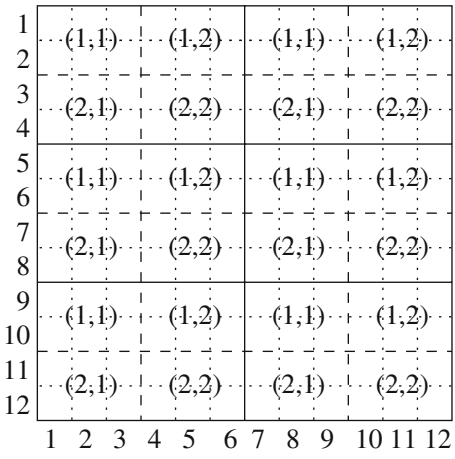


Fig. 7.4 Illustration of a checkerboard distribution for a 12×12 matrix. The *tuples* denote the position of the processors in the processor mesh owning the data block

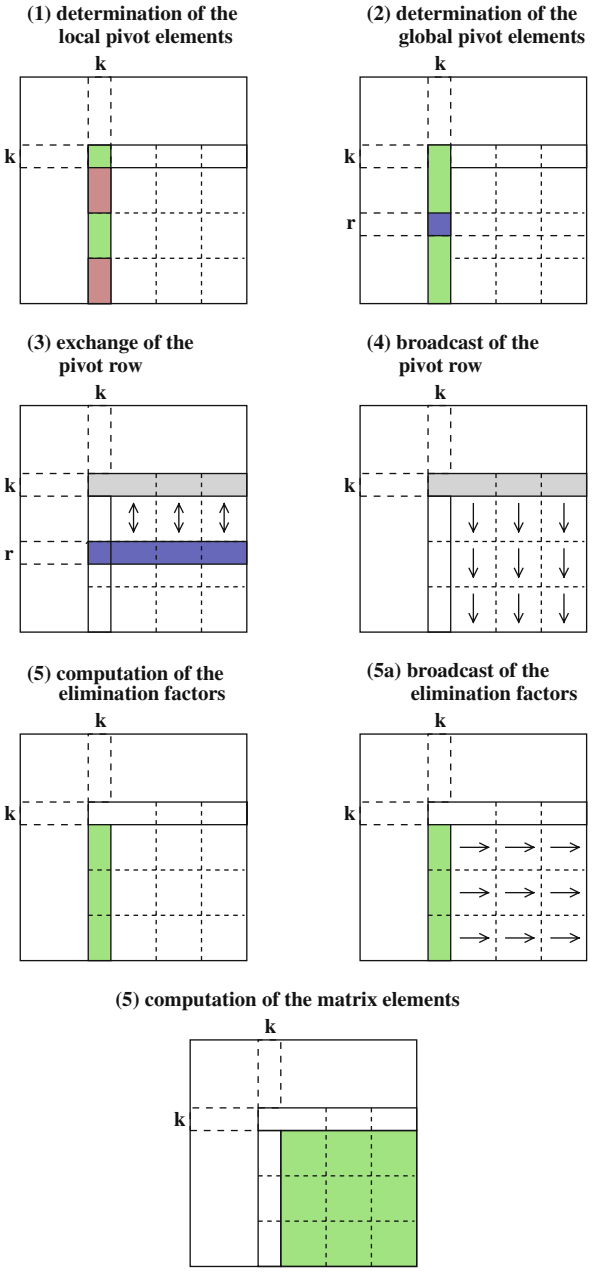


Fig. 7.5 Computation phases of the Gaussian elimination with checkerboard distribution

This figure will be printed in b/w

1. **Determination of the local pivot element:** Since column k is distributed across the processors of column group $Co(k)$, only these processors determine the element with the largest absolute value for row $\geq k$ within their local elements of column k.
2. **Determination of the global pivot element:** The processors in group $Co(k)$ perform a single-accumulation operation within this group, for which each processor in the group provides its local pivot element from phase 1. The reduction operation is the maximum operation also determining the index of the pivot row (and not the number of the owning processor as before). The root processor of the single-accumulation operation is the processor owning the element $a_{kk}^{(k)}$. After the single-accumulation, the root processor knows the pivot element $a_{rk}^{(k)}$ and its row index. This information is sent to all other processors.
3. **Exchange of the pivot row:** The pivot row r containing the pivot element $a_{rk}^{(k)}$ is distributed across row group $Ro(r)$. Row k is distributed across the row group $Ro(k)$, which may be different from $Ro(r)$. If $Ro(r) = Ro(k)$, the processors of $Ro(k)$ exchange the elements of the rows k and r locally within the columns they own. If $Ro(r) \neq Ro(k)$, each processor in $Ro(k)$ sends its part of row k to the corresponding processor in $Ro(r)$; this is the unique processor which belongs to the same column group.
4. **Distribution of the pivot row:** The pivot row is needed for the recalculation of matrix A, but each processor needs only those elements with column indices for which it owns elements. Therefore, each processor in $Ro(r)$ performs a group-oriented single-broadcast operation within its column group sending its part of the pivot row to the other processors.
5. **Computation of the elimination factors:** The processors of column group $Co(k)$ locally compute the elimination factors l_{ik} for their elements i of column k according to Formula (7.2).

5a. **Distribution of the elimination factors:** The elimination factors l_{ik} are needed by all processors in the row group $Ro(i)$. Since the elements of row i are distributed across the row group $Ro(i)$, each processor of column group $Co(k)$ performs a group-oriented single-broadcast operation in its row group $Ro(i)$ to broadcast its elimination factors l_{ik} within this row group.

6. **Computation of the matrix elements:** Each processor locally computes the elements of $A^{(k+1)}$ and $b^{(k+1)}$ using its elements of $A^{(k)}$ and $b^{(k)}$ according to Formulas (7.3) and (7.4).

The backward substitution for computing the n elements of the result vector x is done in n consecutive steps where each step consists of the following computations:

1. Each processor of the row group $Ro(k)$ computes that part of the sum $\sum_{j=k+1}^{n} a_{kj}^{(n)} x_j$ which contains its local elements of row k.
2. The entire sum $\sum_{j=k+1}^{n} a_{kj}^{(n)} x_j$ is determined by the processors of row group $Ro(k)$ by a group-oriented single-accumulation operation with the processor P_q as root which stores the element $a_{kk}^{(n)}$. Addition is used as reduction operation.

3. Processor P_q computes the value of x_k according to Formula (7.5).
4. Processor P_q sends the value of x_k to all other processors by a single-broadcast operation.

A pseudocode for an SPMD program in C notation with MPI operations implementing the Gaussian elimination with checkerboard distribution of matrix A is given in Fig. 7.6. The computations correspond to those given in the pseudocode for the row-cyclic distribution in Fig. 7.2, but the pseudocode additionally uses several functions organizing the computations on the groups of processors. The functions `Co(k)` and `Ro(k)` denote the column or row groups owning column k or row k, respectively. The function `member(me,G)` determines whether processor `me` belongs to group `G`. The function `grp_leader()` determines the first processor in a group. The functions `Cop(q)` and `Rop(q)` determine the column or row group, respectively, to which a processor q belongs. The function `rank(q,G)` returns the local processor number (rank) of a processor in a group `G`.

1. **Determination of the local pivot element:** The determination of the local pivot element is performed only by the processors in column group `Co(k)`.
2. **Determination of the global pivot element:** The global pivot element is again computed by an `MPI_MAXLOC` reduction operation, but in contrast to Fig. 7.2 the index of the row of the pivot element is calculated and not the processor number owning the pivot element. The reason is that all processors which own a part of the pivot row need to know that some of their data belongs to the current pivot row; this information is used in further communication.
3. **Exchange of the pivot row:** For the exchange and distribution of the pivot row r, the cases `Ro(k)` $==$ `Ro(r)` and `Ro(k)` $!=$ `Ro(r)` are distinguished.
   - When the pivot row and the row k are stored by the same row group, each processor of this group exchanges its data elements of row k and row r locally using the function `exchange_row_loc()` and copies the elements of the pivot row (now row k) into the buffer `buf` using the function `copy_row_loc()`. Only the elements in column k or higher are considered.
   - When the pivot row and the row k are stored by different row groups, communication is required for the exchange of the pivot row. The function `compute_partner(Ro(r),me)` computes the communication partner for the calling processor `me`, which is the processor $q \in Ro(r)$ belonging to the same column group as `me`. The function `compute_size(n,k,Ro(k))` computes the number of elements of the pivot row, which is stored for the calling processor in columns greater than k; this number depends on the size of the row group `Ro(k)`, the block size, and the position k. The same function is used later to determine the number of elimination factors to be communicated.
4. **Distribution of the pivot row:** For the distribution of the pivot row r, a processor takes part in a single-broadcast operation in its column group. The roots of the broadcast operation performed in parallel are the processors $q \in$ `Ro(r)`. The participants of a broadcast are the processors $q' \in$ `Cop(q)`, either as root when $q' \in$ `Ro(r)` or as recipient otherwise.

```
double * gauss_double_cyclic (double **a, double *b)
{
 double *x, *buf, *elim_buf;
 int i,j,k,r,q, ql, size, buf_size, elim_size, psz;
 struct { double val; int pvtline; } z,y;
 MPI_Status status;

 x = (double *) malloc(n * sizeof(double));
 buf = (double *) malloc((n+1) * sizeof(double));
 elim_buf = (double *) malloc((n+1) * sizeof(double));
 for (k=0; k<n-1; k++) {
   if (member(me, Co(k))) {
    r = max_col_loc(a,k);
    z.pvtline = r; z.val = fabs(a[r][k]);
    MPI_Reduce(&z,&y,1,MPI_DOUBLE_INT,MPI_MAXLOC,
               grp_leader(Co(k)),comm(Co(k)));
   }
   MPI_Bcast(&y,1,MPI_DOUBLE_INT,grp_leader(Co(k)),MPI_COMM_WORLD);
   r = y.pvtline;
   if(Ro(k) == Ro(r)){
            /*pivot row and row k are in the same row group */
     if (member(me, Ro(k))) {
        if (r != k)  exchange_row_loc(a,b,r,k);
        copy_row_loc(a,b,k,buf); }  }
   else  /* pivot row and row k are in different row groups */
     if (member(me, Ro(k))) {
        copy_row_loc(a,b,k,buf);
        q = compute_partner(Ro(r),me);
        psz = compute_size(n,k,Ro(k));
        MPI_Send(buf+k,psz,MPI_DOUBLE,q,tag,MPI_COMM_WORLD); }
     else  if (member(me,Ro(r))) {
       /* executing processor contains a part of the pivot row */
       q = compute_partner(Ro(k),me);
       psz = compute_size(n,k,Ro(r));
       MPI_Recv(buf+k,psz,MPI_DOUBLE,q,tag,MPI_COMM_WORLD,&status);
       exchange_row_buf(a,b,r,buf,k);
     }
   for (q=0; q<p; q++) /* broadcast of pivot row */
     if (member(q,Ro(r)) && member(me,Cop(q))) {
       ql = rank(q,Cop(q)); buf_size = compute_size(n,k,Ro(k));
       MPI_Bcast(buf+k,buf_size,MPI_DOUBLE,ql,comm(Cop(q)));}
   if ((Ro(k) != Ro(r)) && (member(me,Ro(k)))
     copy_row_loc(a,b,buf,k);
   if  (member(me,Co(k))) elim_buf = compute_elim_fact_loc(a,b,k,buf);
   for (q=0; q<p; q++) /* broadcast of elimination factors */
       if (member(q,Co(k)) && member(me,Rop(q))) {
       ql = rank(q,Rop(q)); elim_size = compute_size(n,k,Co(k));
       MPI_Bcast(elim_buf,elim_size,MPI_DOUBLE,ql,comm(Rop(q))); }
   compute_local_entries(a,b,k,elim_buf,buf); }
 backward_substitution(a,b,x);
 return x;
}
```

Fig. 7.6 Program of the Gaussian elimination with checkerboard distribution

5. **Computation of the elimination factors:** The function `compute_elim_fact_loc()` is used to compute the elimination factors l_{ik} for all elements a_{ik} owned by the processor. The elimination factors are stored in buffer `elim_buf`.

5a. **Distribution of the elimination factors:** A single-broadcast operation is used to send the elimination factors to all processors in the same row group `Rop(q)`; the corresponding communicator `comm(Rop(q))` is used. The number (rank) of the root processor q for this broadcast operation in a group G is determined by the function `rank(q,G)`.

6. **Computation of the matrix elements:** The computation of the matrix elements by `compute_local_entries()` and the backward substitution performed by `backward_substitution()` are similar to the pseudocode in Fig. 7.2.

The main differences to the program in Fig. 7.2 are that more communication is required and that almost all collective communication operations are performed on subgroups of the set of processors and not on the entire set of processors.

7.1.4 Analysis of the Parallel Execution Time

The analysis of the parallel execution time of the Gaussian elimination uses functions expressing the computation and communication times depending on the characteristics of the parallel machine, see also Sect. 4.4. The function describing the parallel execution time of the program in Fig. 7.6 additionally contains the parameters p_1, p_2, b_1, and b_2 of the parameterized data distribution in Formula (7.7). In the following, we model the parallel execution time of the Gaussian elimination with checkerboard distribution, neglecting pivoting and backward substitution for simplicity, see also [147]. These are the phases 4, 5, 5a, and 6 of the Gaussian elimination. For the derivation of functions reflecting the parallel execution time, these four SPMD computation phases can be considered separately, since there is a barrier synchronization after each phase.

For a communication phase, a formula describing the time of a collective communication operation is used which describes the communication time as a function of the number of processors and the message size. For the Gaussian elimination (without pivoting), the phases 4 and 5*a* implement communication with a single-broadcast operation. The communication time for a single-broadcast with p processors and message size m is denoted as $T_{sb}(p, m)$. We assume that independent communication operations on disjoint processor sets can be done in parallel. The values for p and m have to be determined for the specific situation. These parameters depend on the data distribution and the corresponding sizes of row and column groups as well as on the step k, $k = 1, \ldots, n$, of the Gaussian elimination, since messages get smaller for increasing k.

Also, the modeling of the computation times of phases 5 and 6 depends on the step number k, since less elimination factors or matrix elements have to be computed for increasing k and thus the number of arithmetic operations decreases with increasing k. The time for an arithmetic operation is denoted as t_{op}. Since the processors

perform an SPMD program, the processor computing the most arithmetic operations determines the computation time for the specific computation phase. The following modeling of communication and computation times for one step k uses the index sets

$$I_q = \{(i, j) \in \{1 \ldots n\} \times \{1 \ldots n\} \mid P_q \text{ owns } a_{ij}\} ,$$

which contain the indices of the matrix elements stored locally in the memory of processor P_q:

- The broadcasting of the pivot row k in phase 4 of step k sends the elements of row k with column index $\geq k$ to the processors needing the data for computations in rows $\geq k$. Since the pivot row is distributed across the processors of the row group $Ro(k)$, all the processors of $Ro(k)$ send their part of row k. The amount of data sent by one processor $q \in Ro(k)$ is the number of elements of row k with column indices $\geq k$ (i.e., with indices $((k, k), \ldots, (k, n))$) owned by processor q. This is the number

$$N_q^{\text{row}\geq k} := \#\{(k, j) \in I_q | j \geq k\} . \tag{7.8}$$

  (The symbol $\#X$ for a set X denotes the number of elements of this set X.) The processor $q \in Ro(k)$ sends its data to those processors owning elements in the rows with row index $\geq k$ which have the same column indices as the elements of processor q. These are the processors in the column group $Cop(q)$ of the processor q and thus these processors are the recipients of the single-broadcast operation of processor q. Since all column groups of the processors $q \in Ro(k)$ are disjoint, the broadcast operation can be done in parallel and the communication time is

$$\max_{q \in Ro(k)} T_{sb}(\#Cop(q), N_q^{\text{row}\geq k}) .$$

- In phase 5 of step k, the elimination factors using the elements $a_{kk}^{(k)}$ and the elements $a_{ik}^{(k)}$ for $i > k$ are computed by the processors owning these elements of column k, i.e., by the processors $q \in Co(k)$, according to Formula (7.2). Each of the processors computes the elimination factors of its part, which are

$$N_q^{\text{col}>k} := \#\{(i, k) \in I_q | i > k\} \tag{7.9}$$

  elimination factors for processor $q \in Co(k)$. Since the computations are done in parallel, this results in the computation time

$$\max_{q \in Co(k)} N_q^{\text{col}>k} \cdot t_{op} .$$

- In phase 5a the elimination factors are sent to all processors which recalculate the matrix elements with indices $(i, j), i > k, j > k$. Since the elimination

factors $l_{ik}^{(k)}$, $l = k+1, \ldots, n$, are needed within the same row i, a row-oriented single-broadcast operation is used to send the data to the processors owning parts of row i. A processor $q \in Co(k)$ sends its data to the processors in its row group $Rop(q)$. These are the data elements computed in the previous phase, i.e., $N_q^{\text{col}>k}$ data elements, and the communication time is

$$\max_{q \in Co(k)} T_{sb}(\#Rop(q), N_q^{\text{col}>k}) \ .$$

- In phase 6 of step k, all matrix elements in the lower right rectangular area are recalculated. Each processor q recalculates the entries it owns; these are the number of elements per column for rows with indices $> k$ (i.e., $N_q^{\text{col}>k}$) multiplied by the number of elements per row for columns with indices $> k$ (i.e., $N_q^{\text{row}>k}$). Since two arithmetic operations are performed for one entry according to Formula (7.4), the computation time is

$$\max_{q \in P} N_q^{\text{col}>k} \cdot N_q^{\text{row}>k} \cdot 2t_{op} \ .$$

In total, the parallel execution for all phases and all steps is

$$\begin{aligned} T(n, p) = \sum_{k=1}^{n-1} \Big\{ & \max_{q \in Ro(k)} T_{sb}(\#Cop(q), N_q^{\text{row}\geq k}) \\ & + \max_{q \in Co(k)} N_q^{\text{col}>k} \cdot t_{op} \\ & + \max_{q \in Co(k)} T_{sb}(\#Rop(q), N_q^{\text{col}>k}) \\ & + \max_{q \in P} N_q^{\text{col}>k} \cdot N_q^{\text{row}>k} \cdot 2t_{op} \Big\}. \end{aligned} \tag{7.10}$$

This parallel execution time can be expressed in terms of the parameters of the data distribution $((p_1, b_1), (p_2, b_2))$, the problem size n, and the step number k by estimating the sizes of messages and the number of arithmetic operations. For the estimation, larger blocks of data, called **superblocks**, are considered. Superblocks consist of $p_1 \times p_2$ consecutive blocks of size $b_1 \times b_2$, i.e., it has $p_1 b_1$ rows and $p_2 b_2$ columns. There are $\left\lceil \frac{n}{p_1 b_1} \right\rceil$ superblocks in the row direction and $\left\lceil \frac{n}{p_2 b_2} \right\rceil$ in the column direction. Each of the p processors owns one data block of size $b_1 \times b_2$ of a superblock. The two-dimensional matrix A is covered by these superblocks and from this covering, it can be estimated how many elements of smaller matrices $A^{(k)}$ are owned by a specific processor.

The number of elements owned by a processor q in row k for column indices $\geq k$ can be estimated by

$$N_q^{\text{row}\geq k} \leq \left\lceil \frac{n-k+1}{p_2 b_2} \right\rceil b_2 \leq \left(\frac{n-k+1}{p_2 b_2} + 1 \right) b_2 = \frac{n-k+1}{p_2} + b_2, \tag{7.11}$$

where $\left\lceil \frac{n-k+1}{p_2 b_2} \right\rceil$ is the number of superblocks covering row k for column indices $\geq k$, which are $n-k+1$ indices, and b_2 is the number of column elements that each processor of $Ro(k)$ owns in a complete superblock. For the covering of one row, the number of columns $p_2 b_2$ of a superblock is needed. Analogously, the number of elements owned by a processor q in column k for row indices $> k$ can be estimated by

$$N_q^{\text{col}>k} \leq \left\lceil \frac{n-k}{p_1 b_1} \right\rceil b_1 \leq \left(\frac{n-k}{p_1 b_1} + 1 \right) b_1 = \frac{n-k}{p_1} + b_1, \tag{7.12}$$

where $\left\lceil \frac{n-k}{p_1 b_1} \right\rceil$ is the number of superblocks covering column k for row indices $> k$, which are $n-k$ row indices, and b_1 is the number of row elements that each processor of $Co(k)$ owns in a complete superblock. Using these estimations, the parallel execution time in Formula (7.10) can be approximated by

$$\begin{aligned} T(n,p) \approx \sum_{k=1}^{n-1} \Bigg(& T_{sb}\left(p_1, \frac{n-k+1}{p_2} + b_2\right) + \left(\frac{n-k}{p_1} + b_1\right) \cdot t_{op} \\ & + T_{sb}\left(p_2, \frac{n-k}{p_1} + b_1\right) + \left(\frac{n-k}{p_1} + b_1\right)\left(\frac{n-k}{p_2} + b_2\right) \cdot 2t_{op} \Bigg) . \end{aligned}$$

Suitable parameters leading to a good performance can be derived from this modeling. For the communication time of a single-broadcast operation, we assume a communication time

$$T_{sb}(p,m) = \log p \cdot (\tau + m \cdot t_c)$$

with a startup time τ and a transfer time t_c. This formula models the communication time in many interconnection networks, like a hypercube. Using the summation formula $\sum_{k=1}^{n-1}(n-k+1) = \sum_{k=2}^{n} k = (\sum_{k=1}^{n} k) - 1 = \frac{n(n+1)}{2} - 1$ the communication time in phase 4 results in

$$\begin{aligned} & \sum_{k=1}^{n-1} T_{sb}\left(p_1, \frac{n-k+1}{p_2} + b_2\right) \\ & = \sum_{k=1}^{n-1} \log p_1 \left(\left(\frac{n-k+1}{p_2} + b_2 \right) t_c + \tau \right) \\ & = \log p_1 \left(\left(\frac{n(n+1)}{2} - 1 \right) \frac{1}{p_2} t_c + (n-1) b_2 t_c + (n-1)\tau \right) . \end{aligned}$$

For the second and third terms the summation formula $\sum_{k=1}^{n-1}(n-k) = \frac{n(n-1)}{2}$ is used, so that the computation time

$$\sum_{k=1}^{n-1}\left(\frac{n-k}{p_1}+b_1\right)\cdot t_{op}=\left(\frac{n(n-1)}{2p_1}+(n-1)b_1\right)\cdot t_{op}$$

and the communication time

$$\begin{aligned}&\sum_{k=1}^{n-1}T_{sb}\left(p_2,\frac{n-k}{p_1}+b_1\right)\\&=\sum_{k=1}^{n-1}\log p_2\left(\left(\frac{n-k}{p_1}+b_1\right)t_c+\tau\right)\\&=\log p_2\left(\left(\frac{n(n-1)}{2}\right)\frac{1}{p_1}t_c+(n-1)b_1t_c+(n-1)\tau\right)\end{aligned}$$

result. For the last term, the summation formula $\sum_{k=1}^{n-1}\frac{n-k}{p_1}\cdot\frac{n-k}{p_2}=\frac{1}{p}\sum_{k=1}^{n-1}k^2=\frac{1}{p}\frac{n(n-1)(2n-1)}{6}$ is used. The total parallel execution time is

$$\begin{aligned}T(n,p)=&\log p_1\left(\left(\frac{n(n+1)}{2}-1\right)\frac{t_c}{p_2}+(n-1)b_2t_c+(n-1)\tau\right)\\&+\left(\frac{n(n-1)}{2}\frac{1}{p_1}+(n-1)b_1\right)t_{op}\\&+\log p_2\left(\left(\frac{n(n-1)}{2}\frac{t_c}{p_1}\right)+(n-1)b_1t_c+(n-1)\tau\right)\\&+\left(\frac{n(n-1)(2n-1)}{6p}+\frac{n(n-1)}{2}\left(\frac{b_2}{p_1}+\frac{b_1}{p_2}\right)+(n-1)b_1b_2\right)2t_{op}.\end{aligned}$$

The block sizes b_i, $1\le b_i\le n/p_i$, for $i=1,2$ are contained in the execution time as factors and, thus, the minimal execution time is achieved for $b_1=b_2=1$. In the resulting formula the terms $(\log p_1+\log p_2)\,((n-1)(\tau+t_c))=\log p\,((n-1)(\tau+t_c))$, $(n-1)\cdot 3t_{op}$, and $\frac{n(n-1)(2n-1)}{3p}\cdot t_{op}$ are independent of the specific choice of p_1 and p_2 and need not be considered. The terms $\frac{n(n-1)}{2}\frac{1}{p_1}t_{op}$ and $\frac{t_c}{p_2}(n-1)\log p_1$ are asymmetric in p_1 and p_2. For simplicity we ignore these terms in the analysis, which is justified since these terms are small compared to the remaining terms; the first term has t_{op} as operand, which is usually small, and the second term with t_c as operand has a factor only linear in n. The remaining terms of the execution time are symmetric in p_1 and p_2 and have constants quadratic in n. Using $p_2=p/p_1$ this time can be expressed as

$$T_S(p_1)=\frac{n(n-1)}{2}\left(\frac{p_1\log p_1}{p}+\frac{\log p-\log p_1}{p_1}\right)t_c+\frac{n(n-1)}{2}\left(\frac{1}{p_1}+\frac{p_1}{p}\right)2t_{op}\,.$$

The first derivation is

$$T_S'(p_1) = \frac{n(n-1)}{2}\left(\frac{1}{p\cdot\ln 2} + \frac{\log p_1}{p} - \frac{\log p}{p_1^2} + \frac{\log p_1}{p_1^2} - \frac{1}{p_1^2\cdot\ln 2}\right)t_c + \frac{n(n-1)}{2}\left(\frac{1}{p} - \frac{1}{p_1^2}\right)2t_{op}\ .$$

For $p_1 = \sqrt{p}$ it is $T_S'(p_1) = 0$ since $\frac{1}{p} - \frac{1}{p_1^2} = \frac{1}{p} - \frac{1}{p} = 0$, $\frac{1}{p\ln 2} - \frac{1}{p_1^2\ln 2} = 0$, and $\frac{\log p_1}{p} - \frac{\log p}{p_1^2} + \frac{\log p_1}{p_1^2} = 0$. The second derivation $T''(p_1)$ is positive for $p_1 = \sqrt{p}$ and, thus, there is a minimum at $p_1 = p_2 = \sqrt{p}$.

In summary, the analysis of the most influential parts of the parallel execution time of the Gaussian elimination has shown that $p_1 = p_2 = \sqrt{p}$, $b_1 = b_2 = 1$ is the best choice. For an implementation, the values for p_1 and p_2 have to be adapted to integer values.

7.2 Direct Methods for Linear Systems with Banded Structure

Large linear systems with banded structure often arise when discretizing partial differential equations. The coefficient matrix of a banded system is sparse with non-zero elements in the main diagonal of the matrix and a few further diagonals. As a motivation, we first present the discretization of a two-dimensional Poisson equation resulting in such a banded system in Sect. 7.2.1. In Sect. 7.2.2, the solution methods recursive doubling and cyclic reduction are applied to the solution of tridiagonal systems, i.e., banded systems with only three non-zero diagonals, and the parallel implementation is discussed. General banded matrices are treated with cyclic reduction in Sect. 7.2.3 and the discretized Poisson equation is used as an example in Sect. 7.2.4.

7.2.1 Discretization of the Poisson Equation

As a typical example of an elliptic partial differential equation we consider the Poisson equation with Dirichlet boundary conditions. This equation is often called the **model problem** since its structure is simple but the numerical solution is very similar to many other more complicated partial differential equations, see [60, 79, 166]. The two-dimensional Poisson equation has the form

$$-\Delta u(x,y) = f(x,y) \quad \text{for all } (x,y) \in \Omega \tag{7.13}$$

with domain $\Omega \subset \mathbb{R}^2$.

The function $u : \mathbb{R}^2 \to \mathbb{R}$ is the unknown solution function and the function $f : \mathbb{R}^2 \to \mathbb{R}$ is the right-hand side, which is continuous in Ω and its boundary. The operator Δ is the two-dimensional **Laplace operator**

$$\Delta = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}$$

containing the second partial derivatives with respect to x or y. ($\partial/\partial x$ and $\partial/\partial y$ denote the first partial derivatives with respect to x or y, and $\partial^2/\partial x^2$ and $\partial^2/\partial y^2$ denote the second partial derivatives with respect to x or y, respectively.) Using this notation, the Poisson equation (7.13) can also be written as

$$-\frac{\partial^2 u}{\partial x^2} - \frac{\partial^2 u}{\partial y^2} = f(x, y) \, .$$

The model problem (7.13) uses the unit square $\Omega = (0, 1) \times (0, 1)$ and assumes a Dirichlet boundary condition

$$u(x, y) = \varphi(x, y) \quad \text{for all } (x, y) \in \partial\Omega \, , \tag{7.14}$$

where φ is a given function and $\partial\Omega$ is the boundary of domain Ω, which is $\partial\Omega = \{(x, y) \mid 0 \leq x \leq 1, y = 0 \text{ or } y = 1\} \cup \{(x, y) \mid 0 \leq y \leq 1, x = 0 \text{ or } x = 1\}$. The boundary condition uniquely determines the solution u of the model problem. Figure 7.7 (left) illustrates the domain and the boundary of the model problem.
An example of the Poisson equation from electrostatics is the equation

$$\Delta u = -\frac{\rho}{\varepsilon_0} \, ,$$

where ρ is the charge density, ε_0 is a constant, and u is the unknown potential to be determined [97].

For the numerical solution of equation $-\Delta u(x, y) = f(x, y)$, the method of finite differences can be used, which is based on a discretization of the domain $\Omega \cup \partial\Omega$

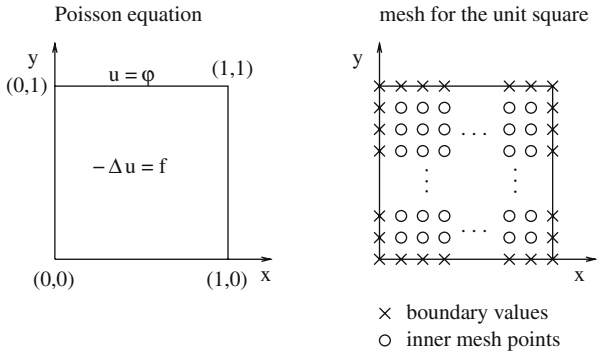


Fig. 7.7 *Left*: Poisson equation with Dirichlet boundary condition on the unit square $\Omega = (0, 1) \times (0, 1)$. *Right*: The numerical solution discretizes the Poisson equation on a mesh with equidistant mesh points with distance $1/(N + 1)$. The mesh has N^2 inner mesh points and additional mesh points on the boundary

in both directions. The discretization is given by a regular mesh with $N + 2$ mesh points in x-direction and in y-direction, where N points are in the inner part and 2 points are on the boundary. The distance between points in the x- or y-direction is $h = \frac{1}{N+1}$. The mesh points are

$$(x_i, y_j) = (ih, jh) \text{ for } i, j = 0, 1, \ldots, N + 1 \ .$$

The points on the boundary are the points with $x_0 = 0, y_0 = 0, x_{N+1} = 1$, or $y_{N+1} = 1$. The unknown solution function u is determined at the points (x_i, y_j) of this mesh, which means that values $u_{ij} := u(x_i, y_j)$ for $i, j = 0, 1, \ldots, N + 1$ are to be found.

For the inner part of the mesh, these values are determined by solving a linear equation system with N^2 equations which is based on the Poisson equation in the following way. For each mesh point (x_i, y_j), $i, j = 1, \ldots, N$, a Taylor expansion is used for the x or y-direction. The Taylor expansion in x-direction is

$$\begin{aligned} u(x_i + h, y_j) &= u(x_i, y_j) + h \cdot u_x(x_i, y_j) + \frac{h^2}{2} u_{xx}(x_i, y_j) \\ &\quad + \frac{h^3}{6} u_{xxx}(x_i, y_j) + \mathrm{O}(h^4) \ , \\ u(x_i - h, y_j) &= u(x_i, y_j) - h \cdot u_x(x_i, y_j) + \frac{h^2}{2} u_{xx}(x_i, y_j) \\ &\quad - \frac{h^3}{6} u_{xxx}(x_i, y_j) + \mathrm{O}(h^4) \ , \end{aligned}$$

where u_x denotes the partial derivative in x-direction (i.e., $u_x = \partial u / \partial x$) and u_{xx} denotes the second partial derivative in x-direction (i.e., $u_{xx} = \partial^2 u / \partial x^2$). Adding these two Taylor expansions results in

$$u(x_i + h, y_j) + u(x_i - h, y_j) = 2u(x_i, y_j) + h^2 u_{xx}(x_i, y_j) + O(h^4) \ .$$

Analogously, the Taylor expansion for the y-direction can be used to get

$$u(x_i, y_j + h) + u(x_i, y_j - h) = 2u(x_i, y_j) + h^2 u_{yy}(x_i, y_j) + O(h^4) \ .$$

From the last two equations, an approximation for the Laplace operator $\Delta u = u_{xx} + u_{yy}$ at the mesh points can be derived

$$\Delta u(x_i, y_j) = -\frac{1}{h^2}(4u_{ij} - u_{i+1,j} - u_{i-1,j} - u_{i,j+1} - u_{i,j-1}) \ ,$$

where the higher order terms $O(h^4)$ are neglected. This approximation uses the mesh point (x_i, y_j) itself and its four neighbor points; see Fig. 7.8. This pattern is known as **five-point stencil**. Using the approximation of Δu and the notation $f_{ij} := f(x_i, y_j)$

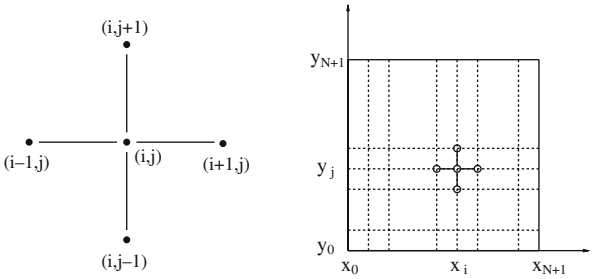


Fig. 7.8 Five-point stencil resulting from the discretization of the Laplace operator with a finite difference scheme. The computation at one mesh point uses values at the four neighbor mesh points

for the values of the right-hand side, the **discretized Poisson equation** or **five-point formula** results:

$$\frac{1}{h^2}(4u_{ij} - u_{i+1,j} - u_{i-1,j} - u_{i,j+1} - u_{i,j-1}) = f_{ij} \tag{7.15}$$

for $1 \leq i, j \leq N$. For the points on the boundary, the values of u_{ij} result from the boundary condition (7.14) and are given by

$$u_{ij} = \varphi(x_i, y_j) \tag{7.16}$$

for $i = 0, N+1$ and $j = 0, \ldots, N+1$ or $j = 0, N+1$ and $i = 0, \ldots, N+1$. The inner mesh points which are immediate neighbors of the boundary, i.e., the mesh points with $i = 1$, $i = N$, $j = 1$, or $j = N$, use the boundary values in their five-point stencil; the four mesh points in the corners use two boundary values and all other points use one boundary value. For all points with $i = 1$, $i = N$, $j = 1$, or $j = N$, the values of u_{ij} in the formulas (7.15) are replaced by the values (7.16). For the mesh point (x_1, y_1) for example, the equation

$$\frac{1}{h^2}(4u_{11} - u_{21} - u_{12}) = f_{11} + \frac{1}{h^2}\varphi(0, y_1) + \frac{1}{h^2}\varphi(x_1, 0)$$

results. The five-point formula (7.15) including boundary values represents a linear equation system with N^2 equations, N^2 unknown values, and a coefficient matrix $A \in \mathbb{R}^{N^2 \times N^2}$. In order to write the equation system (7.15) with boundary values (7.16) in matrix form $Az = d$, the N^2 unknowns u_{ij}, $i, j = 1, \ldots, N$, are arranged in row-oriented order in a one-dimensional vector z of size $n = N^2$ which has the form

$$z = (u_{11}, u_{21}, \ldots, u_{N1}, u_{12}, u_{22}, \ldots, u_{N2}, \ldots.., u_{1N}, u_{2N}, \ldots, u_{NN}) \, .$$

The mapping of values u_{ij} to vector elements z_k is

$$z_k := u_{ij} \text{ with } k = i + (j-1)N \text{ for } i, j = 1, \ldots, N \, .$$

Using the vector z, the five-point formula has the form

$$\frac{1}{h^2}\left(4z_{i+(j-1)N} - z_{i+1+(j-1)N} - z_{i-1+(j-1)N} - z_{i+jN} - z_{i+(j-2)N}\right) = d_{i+(j-1)N}$$

with $d_{i+(j-1)N} = f_{ij}$ and a corresponding mapping of the values f_{ij} to a one-dimensional vector d. Replacing the indices by $k = 1, \ldots, n$ with $k = i + (j-1)N$ results in

$$\frac{1}{h^2}\left(4z_k - z_{k+1} - z_{k-1} - z_{k+N} - z_{k-N}\right) = d_k \ . \tag{7.17}$$

Thus, the entries in row k of the coefficient matrix contain five entries which are $a_{kk} = 4$ and $a_{k,k+1} = a_{k,k-1} = a_{k,k+N} = a_{k,k-N} = -1$.

The building of the vector d and the coefficient matrix $A = (a_{ij})$, $i, j = 1, \ldots, N^2$, can be performed by the following algorithm, see [79]. The loops over i and j, $i, j = 1, \ldots, N$, visit the mesh points (i, j) and build one row of the matrix A of size $N^2 \times N^2$. When (i, j) is an inner point of the mesh, i.e., $i, j \neq 1, N$, the corresponding row of A contains five elements at the position $k, k+1, k-1, k+N, k-N$ for $k = i + (j-1)N$. When (i, j) is at the boundary of the inner part, i.e., $i = 1$, $j = 1$, $i = N$, or $j = N$, the boundary values for φ are used.

```
/* Algorithm for building the matrix A and the vector d */
Initialize all entries of A with 0;
for (j = 1; j <= N; j++)
    for (i = 1; i <= N; i++) {
        /* Build d_k and row k of A with k = i + (j - 1)N */
        k = i + (j - 1) · N;
        a_{k,k} = 4/h^2;
        d_k = f_{ij};
        if (i > 1) a_{k,k-1} = -1/h^2 else d_k = d_k + 1/h^2 φ(0, y_j);
        if (i < N) a_{k,k+1} = -1/h^2 else d_k = d_k + 1/h^2 φ(1, y_j);
        if (j > 1) a_{k,k-N} = -1/h^2 else d_k = d_k + 1/h^2 φ(x_i, 0);
        if (j < N) a_{k,k+N} = -1/h^2 else d_k = d_k + 1/h^2 φ(x_i, 1);
    }
```

The linear equation system resulting from this algorithm has the structure

$$\frac{1}{h^2}\begin{pmatrix} B & -I & & 0 \\ -I & B & \ddots & \\ & \ddots & \ddots & -I \\ 0 & & -I & B \end{pmatrix} \cdot z = d \ , \tag{7.18}$$

where I denotes the $N \times N$ unit matrix, which has the value 1 in the diagonal elements and the value 0 in all other entries. The matrix B has the structure

$$B = \begin{pmatrix} 4 & -1 & & 0 \\ -1 & 4 & \ddots & \\ & \ddots & \ddots & -1 \\ 0 & & -1 & 4 \end{pmatrix}. \tag{7.19}$$

Figure 7.9 illustrates the two-dimensional mesh with five-point stencil (above) and the sparsity structure of the corresponding coefficient matrix A of Formula (7.17).

In summary, Formulas (7.15) and (7.17) represent a linear equation system with a sparse coefficient matrix, which has non-zero elements in the main diagonal and its direct neighbors as well as in the diagonals in distance N. Thus, the linear equation system resulting from the Poisson equation has a banded structure, which should be exploited when solving the system. In the following, we present solution methods for linear equation systems with banded structure and start the description with tridiagonal systems. These systems have only three non-zero diagonals in the main diagonal and its two neighbors. A tridiagonal system results, for example, when discretizing the one-dimensional Poisson equation.

7.2.2 Tridiagonal Systems

For the solution of a linear equation system $Ax = y$ with a banded or tridiagonal coefficient matrix $A \in \mathbb{R}^{n \times n}$, specific solution methods can exploit the sparse matrix structure. A matrix $A = (a_{ij})_{i,j=1,\ldots,n} \in \mathbb{R}^{n \times n}$ is called banded when its structure takes the form of a band of non-zero elements around the principal diagonal. More precisely, this means a matrix A is a **banded matrix** if there exists $r \in \mathbb{N}$, $r \leq n$, with

$$a_{ij} = 0 \text{ for } |i - j| > r \ .$$

The number r is called the **semi-bandwidth** of A. For $r = 1$ a banded matrix is called **tridiagonal matrix**. We first consider the solution of tridiagonal systems which are linear equation systems with tridiagonal coefficient matrix.

7.2.2.1 Gaussian Elimination for Tridiagonal Systems

For the solution of a linear equation system $Ax = y$ with tridiagonal matrix A, the Gaussian elimination can be used. Step k of the forward elimination (without pivoting) results in the following computations, see also Sect. 7.1:

1. Compute $l_{ik} := a_{ik}^{(k)} / a_{kk}^{(k)}$ for $i = k + 1, \ldots, n$.
2. Subtract l_{ik} times the kth row from the rows $i = k + 1, \ldots, n$, i.e., compute

$$a_{ij}^{(k+1)} = a_{ij}^{(k)} - l_{ik} \cdot a_{kj}^{(k)} \quad \text{for } k \leq j \leq n \text{ and } k < i \leq n \ .$$

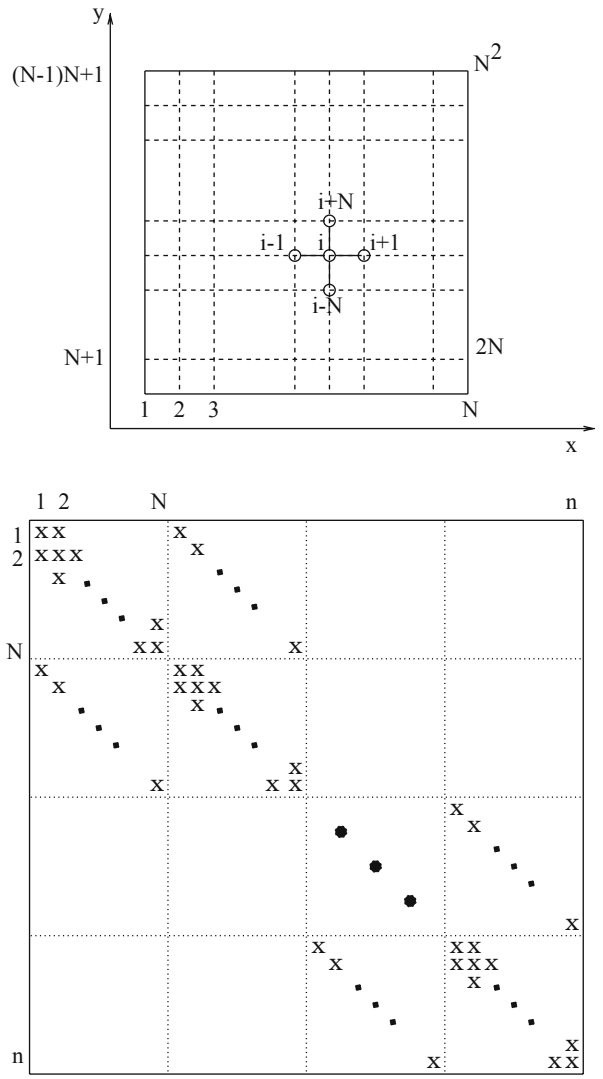


Fig. 7.9 Rectangular mesh in the x–y plane of size $N \times N$ and the $n \times n$ coefficient matrix with $n = N^2$ of the corresponding linear equation system of the five-point formula. The sparsity structure of the matrix corresponds to the adjacency relation of the mesh points. The mesh can be considered as adjacency graph of the non-zero elements of the matrix

The vector y is changed analogously.

Because of the tridiagonal structure of A, all matrix elements a_{ik} with $i \geq k+2$ are zero elements, i.e., $a_{ik} = 0$. Thus, in each step k of the Gaussian elimination only one elimination factor $l_{k+1} := l_{k+1,k}$ and only one row with only one new element have to be computed. Using the notation

$$A = \begin{pmatrix} b_1 & c_1 & & & 0 \\ a_2 & b_2 & c_2 & & \\ & a_3 & b_3 & \ddots & \\ & & \ddots & \ddots & c_{n-1} \\ 0 & & & a_n & b_n \end{pmatrix} \tag{7.20}$$

for the matrix elements and starting with $u_1 = b_1$, these computations are

$$\begin{aligned} l_{k+1} &= a_{k+1}/u_k, \\ u_{k+1} &= b_{k+1} - l_{k+1} \cdot c_k \ . \end{aligned} \tag{7.21}$$

After $n-1$ steps an LU decomposition $A = LU$ of matrix (7.20) with

$$L = \begin{pmatrix} 1 & & & 0 \\ l_2 & 1 & & \\ & \ddots & \ddots & \\ 0 & & l_n & 1 \end{pmatrix} \quad \text{and} \quad U = \begin{pmatrix} u_1 & c_1 & & 0 \\ & \ddots & \ddots & \\ & & u_{n-1} & c_{n-1} \\ 0 & & & u_n \end{pmatrix}$$

results. The right-hand side y is transformed correspondingly according to

$$\tilde{y}_{k+1} = y_{k+1} - l_{k+1} \cdot \tilde{y}_k \ .$$

The solution x is computed from the upper triangular matrix U by a backward substitution, starting with $x_n = \tilde{y}_n/u_n$ and solving the equations $u_i x_i + c_i x_{i+1} = \tilde{y}_i$ one after another resulting in

$$x_i = \frac{\tilde{y}_i}{u_i} - \frac{c_i}{u_i} x_{i+1} \quad \text{for } i = n-1, \ldots, 1 \ .$$

The computational complexity of the Gaussian elimination is reduced to $O(n)$ for tridiagonal systems. However, the elimination phase computing l_k and u_k according to Eq. (7.21) is inherently sequential, since the computation of l_{k+1} depends on u_k and the computation of u_{k+1} depends on l_{k+1}. Thus, in this form the Gaussian elimination or LU decomposition has to be computed sequentially and is not suitable for a parallel implementation.

7.2.2.2 Recursive Doubling for Tridiagonal Systems

An alternative approach for solving a linear equation system with tridiagonal matrix is the method of **recursive doubling** or **cyclic reduction**. The methods of recursive doubling and cyclic reduction also use elimination steps but contain potential parallelism [72, 71]. Both techniques can be applied if the coefficient matrix is either symmetric and positive definite or diagonal dominant [115]. The elimination steps

in both methods are applied to linear equation systems $Ax = y$ with the matrix structure shown in (7.20), i.e.,

$$\begin{array}{rcll} b_1\ x_1 + c_1\ x_2 & = & y_1, & \\ a_i\ x_{i-1} + b_i\ x_i + c_i\ x_{i+1} & = & y_i & \text{for } i = 2, \ldots, n-1, \\ a_n\ x_{n-1} + b_n\ x_n & = & y_n & . \end{array}$$

The method, which was first introduced by Hockney and Golub in [91], uses two equations $i - 1$ and $i + 1$ to eliminate the variables x_{i-1} and x_{i+1} from equation i. This results in a new equivalent equation system with a coefficient matrix with three non-zero diagonals where the diagonals are moved to the outside. Recursive doubling and cyclic reduction can be considered as two implementation variants for the same numerical idea of the method of Hockney and Golub. The implementation of recursive doubling repeats the elimination step, which finally results in a matrix structure in which only the elements in the principal diagonal are non-zero and the solution vector x can be computed easily. Cyclic reduction is a variant of recursive doubling which also eliminates variables using neighboring rows. But in each step the elimination is only applied to half of the equations and, thus, less computations are performed. On the other hand, the computation of the solution vector x requires a substitution phase.

We would like to mention that the terms recursive doubling and cyclic reduction are used in different ways in the literature. Cyclic reduction is sometimes used for the numerical method of Hockney and Golub in both implementation variants, see [60, 115]. On the other hand the term recursive doubling (or full recursive doubling) is sometimes used for a different method, the method of Stone [168]. This method applies the implementation variants sketched above in Eq. (7.21) resulting from the Gaussian elimination, see [61, 173]. In the following, we start the description of recursive doubling for the method of Hockney and Golub according to [61] and [13].

Recursive doubling considers three neighboring equations $i - 1, i, i + 1$ of the equation system $Ax = y$ with coefficient matrix A in the form (7.20) for $i = 3, 4, \ldots, n - 2$. These equations are

$$\begin{array}{rrrrl} a_{i-1}x_{i-2} + b_{i-1}x_{i-1} + c_{i-1}x_i & & & & = y_{i-1}, \\ a_i x_{i-1} & + b_i x_i & + c_i x_{i+1} & & = y_i, \\ & a_{i+1}x_i + b_{i+1}x_{i+1} & + c_{i+1}x_{i+2} & & = y_{i+1}. \end{array}$$

Equation $i - 1$ is used to eliminate x_{i-1} from the ith equation and equation $i + 1$ is used to eliminate x_{i+1} from the ith equation. This is done by reformulating equations $i - 1$ and $i + 1$ to

$$\begin{aligned} x_{i-1} &= \frac{y_{i-1}}{b_{i-1}} - \frac{a_{i-1}}{b_{i-1}}x_{i-2} - \frac{c_{i-1}}{b_{i-1}}x_i\ , \\ x_{i+1} &= \frac{y_{i+1}}{b_{i+1}} - \frac{a_{i+1}}{b_{i+1}}x_i - \frac{c_{i+1}}{b_{i+1}}x_{i+2} \end{aligned}$$

and inserting those descriptions of x_{i-1} and x_{i+1} into equation i. The resulting new equation i is

$$a_i^{(1)} x_{i-2} + b_i^{(1)} x_i + c_i^{(1)} x_{i+2} = y_i^{(1)} \tag{7.22}$$

with coefficients

$$\begin{aligned} a_i^{(1)} &= \alpha_i^{(1)} \cdot a_{i-1}, \\ b_i^{(1)} &= b_i + \alpha_i^{(1)} \cdot c_{i-1} + \beta_i^{(1)} \cdot a_{i+1}, \\ c_i^{(1)} &= \beta_i^{(1)} \cdot c_{i+1}, \\ y_i^{(1)} &= y_i + \alpha_i^{(1)} \cdot y_{i-1} + \beta_i^{(1)} \cdot y_{i+1}, \end{aligned} \tag{7.23}$$

and

$$\begin{aligned} \alpha_i^{(1)} &:= -a_i / b_{i-1} \ , \\ \beta_i^{(1)} &:= -c_i / b_{i+1} \ . \end{aligned}$$

For the special cases $i = 1, 2, n-1, n$, the coefficients are given by

$$\begin{aligned} &b_1^{(1)} = b_1 + \beta_1^{(1)} \cdot a_2, \quad y_1^{(1)} = y_1 + \beta_1^{(1)} \cdot y_2, \\ &b_n^{(1)} = b_n + \alpha_n^{(1)} \cdot c_{n-1}, \; y_n^{(1)} = b_n + \alpha_n^{(1)} \cdot y_{n-1}, \\ &a_1^{(1)} = a_2^{(1)} = 0, \text{ and} \quad c_{n-1}^{(1)} = c_n^{(1)} = 0 \ . \end{aligned}$$

The values for $a_{n-1}^{(1)}$, $a_n^{(1)}$, $b_2^{(1)}$, $b_{n-1}^{(1)}$, $c_1^{(1)}$, $c_2^{(1)}$, $y_2^{(1)}$, and $y_{n-1}^{(1)}$ are defined as in Eq. (7.23). Equation (7.22) forms a linear equation system $A^{(1)}x = y^{(1)}$ with a coefficient matrix

$$A^{(1)} = \begin{pmatrix} b_1^{(1)} & 0 & c_1^{(1)} & & & 0 \\ 0 & b_2^{(1)} & 0 & c_2^{(1)} & & \\ a_3^{(1)} & 0 & b_3^{(1)} & \ddots & \ddots & \\ & a_4^{(1)} & \ddots & \ddots & \ddots & c_{n-2}^{(1)} \\ & & \ddots & \ddots & \ddots & 0 \\ 0 & & & a_n^{(1)} & 0 & b_n^{(1)} \end{pmatrix} .$$

Comparing the structure of $A^{(1)}$ with the structure of A, it can be seen that the diagonals are moved to the outside.

In the next step, this method is applied to the equations $i-2, i, i+2$ of the equation system $A^{(1)}x = y^{(1)}$ for $i = 5, 6, \ldots, n-4$. Equation $i-2$ is used to eliminate x_{i-2} from the ith equation and equation $i+2$ is used to eliminate x_{i+2} from the ith equation. This results in a new ith equation

$$a_i^{(2)} x_{i-4} + b_i^{(2)} x_i + c_i^{(2)} x_{i+4} = y_i^{(2)},$$

which contains the variables x_{i-4}, x_i, and x_{i+4}. The cases $i = 1, \ldots, 4, n-3, \ldots, n$ are treated separately as shown for the first elimination step. Altogether a next equation system $A^{(2)}x = y^{(2)}$ results in which the diagonals are further moved to the outside. The structure of $A^{(2)}$ is

$$A^{(2)} = \begin{pmatrix} b_1^{(2)} & 0 & 0 & 0 & c_1^{(2)} & & & 0 \\ 0 & b_2^{(2)} & & & & c_2^{(2)} & & \\ 0 & & \ddots & & & & \ddots & \\ 0 & & & \ddots & & & & c_{n-4}^{(2)} \\ a_5^{(2)} & & & & \ddots & & & 0 \\ & a_6^{(2)} & & & & \ddots & & 0 \\ & & \ddots & & & & \ddots & 0 \\ 0 & & & a_n^{(2)} & 0 & 0 & 0 & b_n^{(2)} \end{pmatrix}.$$

The following steps of the recursive doubling algorithm apply the same method to the modified equation system of the last step. Step k transfers the side diagonals $2^k - 1$ positions away from the main diagonal, compared to the original coefficient matrix. This is reached by considering equations $i - 2^{k-1}, i, i + 2^{k-1}$:

$$\begin{aligned} a_{i-2^{k-1}}^{(k-1)} x_{i-2^k} + b_{i-2^{k-1}}^{(k-1)} x_{i-2^{k-1}} + c_{i-2^{k-1}}^{(k-1)} x_i &= y_{i-2^{k-1}}^{(k-1)}, \\ a_i^{(k-1)} x_{i-2^{k-1}} + b_i^{(k-1)} x_i + c_i^{(k-1)} x_{i+2^{k-1}} &= y_i^{(k-1)}, \\ a_{i+2^{k-1}}^{(k-1)} x_i + b_{i+2^{k-1}}^{(k-1)} x_{i+2^{k-1}} + c_{i+2^{k-1}}^{(k-1)} x_{i+2^k} &= y_{i+2^{k-1}}^{(k-1)}. \end{aligned}$$

Equation $i - 2^{k-1}$ is used to eliminate $x_{i-2^{k-1}}$ from the ith equation and equation $i + 2^{k-1}$ is used to eliminate $x_{i+2^{k-1}}$ from the ith equation. Again, the elimination is performed by computing the coefficients for the next equation system. These coefficients are

$$\begin{aligned} a_i^{(k)} &= \alpha_i^{(k)} \cdot a_{i-2^{k-1}}^{(k-1)} \quad \text{for } i = 2^k + 1, \ldots, n, \text{ and } a_i^{(k)} = 0 \text{ otherwise}, \\ c_i^{(k)} &= \beta_i^{(k)} \cdot c_{i+2^{k-1}}^{(k-1)} \quad \text{for } i = 1, \ldots, n - 2^k, \text{ and } c_i^{(k)} = 0 \text{ otherwise}, \\ b_i^{(k)} &= \alpha_i^{(k)} \cdot c_{i-2^{k-1}}^{(k-1)} + b_i^{(k-1)} + \beta_i^{(k)} \cdot a_{i+2^{k-1}}^{(k-1)} \quad \text{for } i = 1, \ldots, n, \\ y_i^{(k)} &= \alpha_i^{(k)} \cdot y_{i-2^{k-1}}^{(k-1)} + y_i^{(k-1)} + \beta_i^{(k)} \cdot y_{i+2^{k-1}}^{(k-1)} \quad \text{for } i = 1, \ldots, n \end{aligned} \tag{7.24}$$

with

$$\begin{aligned} \alpha_i^{(k)} &:= -a_i^{(k-1)} / b_{i-2^{k-1}}^{(k-1)} \quad \text{for } i = 2^{k-1} + 1, \ldots, n, \\ \beta_i^{(k)} &:= -c_i^{(k-1)} / b_{i+2^{k-1}}^{(k-1)} \quad \text{for } i = 1, \ldots, n - 2^{k-1}. \end{aligned} \tag{7.25}$$

The modified equation i results by multiplying equation $i - 2^{k-1}$ from step $k-1$ with $\alpha_i^{(k)}$, multiplying equation $i + 2^{k-1}$ from step $k-1$ with $\beta_i^{(k)}$, and adding both to equation i. The resulting ith equation is

$$a_i^{(k)} x_{i-2^k} + b_i^{(k)} x_i + c_i^{(k)} x_{i+2^k} = y_i^{(k)} \tag{7.26}$$

with the coefficients (7.24). The cases $k = 1, 2$ are special cases of this formula. The initialization for $k = 0$ is the following:

$$\begin{aligned} a_i^{(0)} &= a_i && \text{for } i = 2, \ldots, n \ , \\ b_i^{(0)} &= b_i && \text{for } i = 1, \ldots, n \ , \\ c_i^{(0)} &= c_i && \text{for } i = 1, \ldots, n-1 \ , \\ y_i^{(0)} &= y_i && \text{for } i = 1, \ldots, n \ . \end{aligned}$$

and $a_1^{(0)} = 0$, $c_n^{(0)} = 0$. Also, for the steps $k = 0, \ldots, \lceil \log n \rceil$ and $i \in \mathbb{Z} \setminus \{1, \ldots, n\}$ the values

$$\begin{aligned} a_i^{(k)} = c_i^{(k)} = y_i^{(k)} &= 0 \ , \\ b_i^{(k)} &= 1 \ , \\ x_i &= 0 \end{aligned}$$

are set. After $N = \lceil \log n \rceil$ steps, the original matrix A is transformed into a diagonal matrix $A^{(N)}$

$$A^{(N)} = \text{diag}(b_1^{(N)}, \ldots, b_n^{(N)})$$

in which only the main diagonal contains non-zero elements. The solution x of the linear equation system can be directly computed using this matrix and the correspondingly modified vector $y^{(N)}$:

$$x_i = y_i^{(N)} / b_i^{(N)} \text{ for } i = 1, 2, \ldots, n \ .$$

To summarize, the recursive doubling algorithm consists of two main phases:

1. Elimination phase: Compute the values $a_i^{(k)}$, $b_i^{(k)}$, $c_i^{(k)}$, and $y_i^{(k)}$ for $k = 1, \ldots, \lceil \log n \rceil$ and $i = 1, \ldots, n$ according to Eqs. (7.24) and (7.25).
2. Solution phase: Compute $x_i = y_i^{(N)} / b_i^{(N)}$ for $i = 1, \ldots, n$ with $N = \lceil \log n \rceil$.

The first phase consists of $\lceil \log n \rceil$ steps where in each step $O(n)$ values are computed. The sequential asymptotic runtime of the algorithm is therefore $O(n \cdot \log n)$ which is asymptotically slower than the $O(n)$ runtime for the Gaussian elimination approach described earlier. The advantage is that the computations in each step of the elimination and the substitution phase are independent and can be performed in parallel. Figure 7.10 illustrates the computations of the recursive doubling algorithm and the data dependencies between different steps.

7.2.2.3 Cyclic Reduction for Tridiagonal Systems

The recursive doubling algorithm offers a large degree of potential parallelism but has a larger computational complexity than the Gaussian elimination caused by

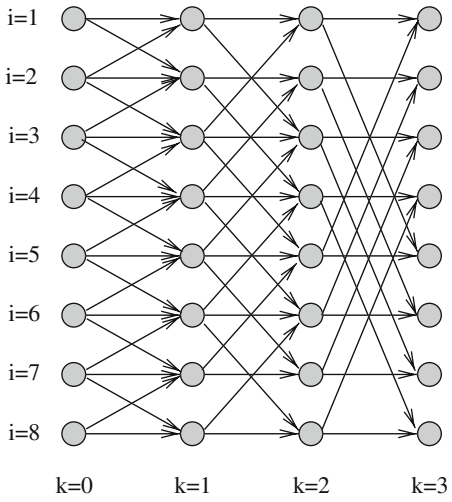


Fig. 7.10 Dependence graph for the computation steps of the recursive doubling algorithm in the case of three computation steps and eight equations. The computations of step k are shown in column k of the illustration. Column k contains one node for each equation i, thus representing the computation of all coefficients needed in step k. Column 0 represents the data of the coefficient matrix of the linear system. An edge from a node i in step k to a node j in step $k+1$ means that the computation at node j needs at least one coefficient computed at node i

computational redundancy. The **cyclic reduction** algorithm is a modification of recursive doubling which reduces the amount of computations to be performed. In each step, half the variables in the equation system are eliminated which means that only half of the values $a_i^{(k)}$, $b_i^{(k)}$, $c_i^{(k)}$, and $y_i^{(k)}$ are computed. A substitution phase is needed to compute the solution vector x. The elimination and the substitution phases of cyclic reduction are described by the following two phases:

1. Elimination phase: For $k = 1, \ldots, \lfloor \log n \rfloor$ compute $a_i^{(k)}$, $b_i^{(k)}$, $c_i^{(k)}$, and $y_i^{(k)}$ with $i = 2^k, \ldots, n$ and step size 2^k. The number of equations of the form (7.26) is reduced by a factor of $1/2$ in each step. In step $k = \lfloor \log n \rfloor$ there is only one equation left for $i = 2^N$ with $N = \lfloor \log n \rfloor$.
2. Substitution phase: For $k = \lfloor \log n \rfloor, \ldots, 0$ compute x_i according to Eq. (7.26) for $i = 2^k, \ldots, n$ with step size 2^{k+1}:

$$x_i = \frac{y_i^{(k)} - a_i^{(k)} \cdot x_{i-2^k} - c_i^{(k)} \cdot x_{i+2^k}}{b_i^{(k)}} \,. \tag{7.27}$$

Figure 7.11 illustrates the computations of the elimination and the substitution phases of cyclic reduction represented by nodes and their dependencies represented by arrows. In each computation step k, $k = 1, \ldots, \lfloor \log n \rfloor$, of the elimination phase,

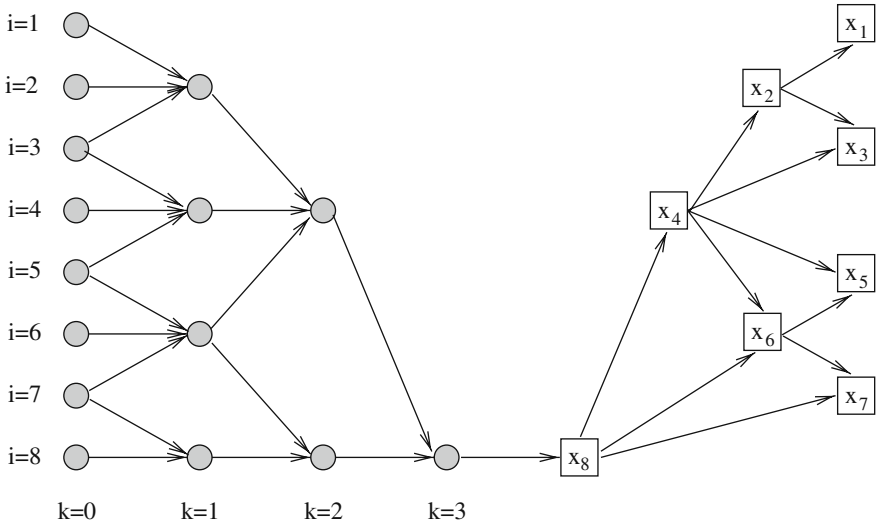


Fig. 7.11 Dependence graph illustrating the dependencies between neighboring computation steps of the cyclic reduction algorithm for the case of three computation steps and eight equations in analogy to the representation in Fig. 7.10. The first four columns represent the computations of the coefficients. The last columns in the graph represent the computation of the solution vector x in the second phase of the cyclic reduction algorithm, see (7.27)

there are $n/2^k$ nodes representing the computations for the coefficients of one equation. This results in

$$\frac{n}{2} + \frac{n}{4} + \frac{n}{8} + \cdots + \frac{n}{2^N} = n \cdot \sum_{i=1}^{\lfloor \log n \rfloor} \frac{1}{2^i} \leq n$$

computation nodes with $N = \lfloor \log n \rfloor$ and, therefore, the execution time of cyclic reduction is $O(n)$. Thus, the computational complexity is the same as for the Gaussian elimination; however, the cyclic reduction offers potential parallelism which can be exploited in a parallel implementation as described in the following.

The computations of the numbers $\alpha_i^{(k)}$, $\beta_i^{(k)}$ require a division by $b_i^{(k)}$ and, thus, cyclic reduction as well as recursive doubling is not possible if any number $b_i^{(k)}$ is zero. This can happen even when the original matrix is invertible and has non-zero diagonal elements or when the Gaussian elimination can be applied without pivoting. However, for many classes of matrices it can be shown that a division by zero is never encountered. Examples are matrices A which are symmetric and positive definite or invertible and diagonally dominant, see [61] or [115] (using the name odd–even reduction). (A matrix A is symmetric if $A = A^T$ and positive definite if $x^T Ax > 0$ for all x. A matrix is diagonally dominant if in each row the absolute value of the diagonal element exceeds the sum of the absolute values of the other elements in the row without the diagonal in the row.)

7.2.2.4 Parallel Implementation of Cyclic Reduction

We consider a parallel algorithm for the cyclic reduction for p processors. For the description of the phases we assume $n = p \cdot q$ for $q \in \mathbb{N}$ and $q = 2^Q$ for $Q \in \mathbb{N}$. Each processor stores a block of rows of size q, i.e., processor P_i stores the rows of A with the numbers $(i-1)q+1, \ldots, i \cdot q$ for $1 \leq i \leq p$. We describe the parallel algorithm with data exchange operations that are needed for an implementation with a distributed address space. As data distribution a row-blockwise distribution of the matrix A is used to reduce the interaction between processors as much as possible. The parallel algorithm for the cyclic reduction comprises three phases: the elimination phase stopping earlier than described above, an additional recursive doubling phase, and a substitution phase.

Phase 1: Parallel reduction of the cyclic reduction in $\log q$ steps: Each processor computes the first $Q = \log q$ steps of the cyclic reduction algorithm, i.e., processor P_i computes for $k = 1, \ldots, Q$ the values

$$a_j^{(k)}, b_j^{(k)}, c_j^{(k)}, y_j^{(k)}$$

for $j = (i-1) \cdot q + 2^k, \ldots, i \cdot q$ with step size 2^k. After each computation step, processor P_i receives four data values from P_{i-1} (if $i > 1$) and from processor P_{i+1} (if $i < n$) computed in the previous step. Since each processor owns a block of rows of size q, no communication with any other processor is required. The size of data to be exchanged with the neighboring processors is a multiple of 4 since four coefficients $(a_j^{(k)}, b_j^{(k)}, c_j^{(k)}, y_j^{(k)})$ are transferred. Only one data block is received per step and so there are at most $2Q$ messages of size 4 for each step.

Phase 2: Parallel recursive doubling for tridiagonal systems of size p: Processor P_i is responsible for the ith equation of the following p-dimensional tridiagonal system

$$\tilde{a}_i \tilde{x}_{i-1} + \tilde{b}_i \tilde{x}_i + \tilde{c}_i \tilde{x}_{i+1} = \tilde{y}_i \qquad \text{for } i = 1, \ldots, p$$

with

$$\left.\begin{array}{rcl} \tilde{a}_i & = & a_{i \cdot q}^{(Q)} \\ \tilde{b}_i & = & b_{i \cdot q}^{(Q)} \\ \tilde{c}_i & = & c_{i \cdot q}^{(Q)} \\ \tilde{y}_i & = & y_{i \cdot q}^{(Q)} \\ \tilde{x}_i & = & x_{i \cdot q} \end{array}\right\} \quad \text{for } i = 1, \ldots, p\ .$$

For the solution of this system, we use recursive doubling. Each processor is assigned one equation. Processor P_i performs $\lceil \log p \rceil$ steps of the recursive doubling algorithm. In step k, $k = 1, \ldots, \lceil \log p \rceil$, processor P_i computes

$$\tilde{a}_i^{(k)}, \tilde{b}_i^{(k)}, \tilde{c}_i^{(k)}, \tilde{y}_i^{(k)}$$

for which the values of

$$\tilde{a}_j^{(k-1)}, \tilde{b}_j^{(k-1)}, \tilde{c}_j^{(k-1)}, \tilde{y}_j^{(k-1)}$$

from the previous step computed by a different processor are required. Thus, there is a communication in each of the $\lceil \log p \rceil$ steps with a message size of four values. After step $N' = \lceil \log p \rceil$ processor P_i computes

$$\tilde{x}_i = \tilde{y}_i^{(N')} / \tilde{b}_i^{(N')}. \tag{7.28}$$

Phase 3: Parallel substitution of cyclic reduction: After the second phase, the values $\tilde{x}_i = x_{i \cdot q}$ are already computed. In this phase, each processor P_i, $i = 1, \ldots, p$, computes the values x_j with $j = (i-1)q + 1, \ldots, iq - 1$ in several steps according to Eq. (7.27). In step k, $k = Q - 1, \ldots, 0$, the elements x_j, $j = 2^k, \ldots, n$, with step size 2^{k+1} are computed. Processor P_i computes x_j with $j \operatorname{div} q + 1 = i$ for which the values $\tilde{x}_{i-1} = x_{(i-1)q}$ and $\tilde{x}_{i+1} = x_{(i+1)q}$ computed by processors P_{i-1} and P_{i+1} are needed. Figure 7.12 illustrates the parallel algorithm for $p = 2$ and $n = 8$.

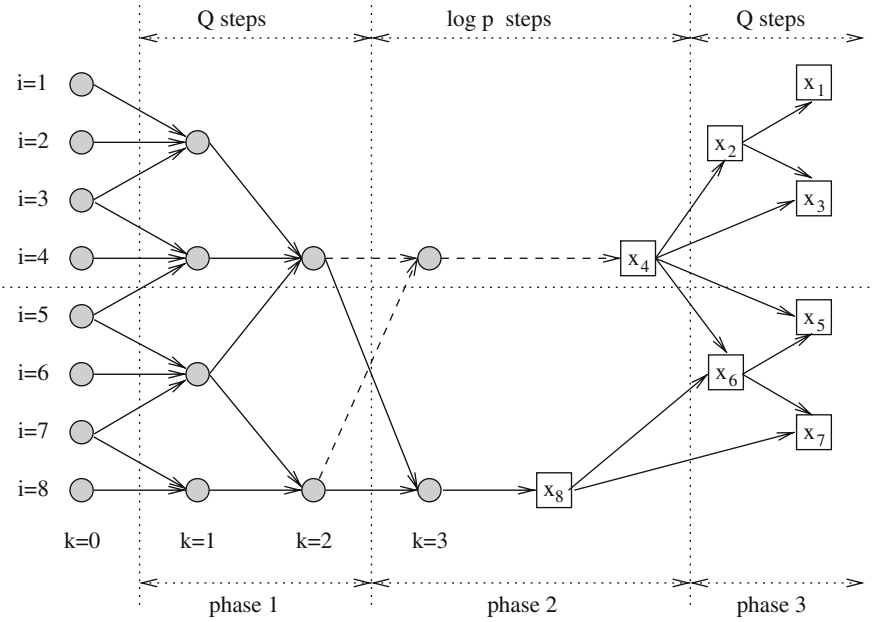


Fig. 7.12 Illustration of the parallel algorithm for the cyclic reduction for $n = 8$ equations and $p = 2$ processors. Each of the processors is responsible for $q = 4$ equations; we have $Q = 2$. The first and the third phases of the computation have $\log q = 2$ steps. The second phase has $\log p = 1$ step. As recursive doubling is used in the second phase, there are more components of the solution to be computed in the second phase compared with the computation shown in Fig. 7.11

7.2.2.5 Parallel Execution Time

The execution time of the parallel algorithm can be modeled by the following runtime functions. Phase 1 executes

$$Q = \log q = \log \frac{n}{p} = \log n - \log p$$

steps where in step k with $1 \leq k \leq Q$ each processor computes at most $q/2^k$ coefficient blocks of 4 values each. Each coefficient block requires 14 arithmetic operations according to Eq. (7.23). The computation time of phase 1 can therefore be estimated as

$$T_1(n, p) = 14 t_{op} \cdot \sum_{k=1}^{Q} \frac{q}{2^k} \leq 14 \frac{n}{p} \cdot t_{op} \ .$$

Moreover, each processor exchanges in each of the Q steps two messages of 4 values each with its two neighboring processors by participating in single transfer operations. Since in each step the transfer operations can be performed by all processors in parallel without interference, the resulting communication time is

$$C_1(n, p) = 2Q \cdot t_{s2s}(4) = 2 \cdot \log \frac{n}{p} \cdot t_{s2s}(4) \ ,$$

where $t_{s2s}(m)$ denotes the time of a single transfer operation with message size m. Phase 2 executes $\lceil \log p \rceil$ steps. In each step, each processor computes 4 coefficients requiring 14 arithmetic operations. Then the value $\tilde{x}_i = x_{i \cdot q}$ is computed according to Eq. (7.28) by a single arithmetic operation. The computation time is therefore

$$T_2(n, p) = 14 \lceil \log p \rceil \cdot t_{op} + t_{op} \ .$$

In each step, each processor sends and receives 4 data values from other processors, leading to a communication time

$$C_2(n, p) = 2 \lceil \log p \rceil \cdot t_{s2s}(4) \ .$$

In each step k of phase 3, $k = 0, \ldots, Q-1$, each processor computes 2^k components of the solution vector according to Eq. (7.27). For each component, five operations are needed. Altogether, each processor computes $\sum_{k=0}^{Q-1} 2^k = 2^Q - 1 = q - 1$ components with one component already computed in phase 2. The resulting computation time is

$$T_3(n, p) = 5 \cdot (q - 1) \cdot t_{op} = 5 \cdot \left(\frac{n}{p} - 1 \right) \cdot t_{op} \ .$$

Moreover, each processor exchanges one data value with each of its neighboring processors; the communication time is therefore

$$C_3(n, p) = 2 \cdot t_{s2s}(1) .$$

The resulting total computation time is

$$\begin{aligned} T(n, p) &= \left(14\frac{n}{p} + 14 \cdot \lceil \log p \rceil + 5\frac{n}{p} - 4 \right) \cdot \ t_{op} \\ &\simeq \left(19\frac{n}{p} + 14 \cdot \log p \right) \cdot \ t_{op} . \end{aligned}$$

The communication overhead is

$$\begin{aligned} C(n, p) &= \left[\ 2 \cdot \log \frac{n}{p} + 2\lceil \log p \rceil \right] \ t_{s2s}(4) + 2 \cdot t_{s2s}(1) \\ &\simeq 2 \cdot \log n \cdot t_{s2s}(4) + 2 \cdot t_{s2s}(1) . \end{aligned}$$

Compared to the sequential algorithm, the parallel implementation leads to a small computational redundancy of $14 \cdot \log p$ operations. The communication overhead increases logarithmically with the number of rows, whereas the computation time increases linearly.

7.2.3 Generalization to Banded Matrices

The cyclic reduction algorithm can be generalized to banded matrices with semi-bandwidth $r > 1$. For the description we assume $n = s \cdot r$. The matrix is represented as a block-tridiagonal matrix of the form

$$\begin{pmatrix} B_1^{(0)} & C_1^{(0)} & & & 0 \\ A_2^{(0)} & B_2^{(0)} & C_2^{(0)} & & \\ & \ddots & \ddots & \ddots & \\ & & A_{s-1}^{(0)} & B_{s-1}^{(0)} & C_{s-1}^{(0)} \\ 0 & & & A_s^{(0)} & B_s^{(0)} \end{pmatrix} \begin{pmatrix} X_1 \\ X_2 \\ \vdots \\ X_{s-1} \\ X_s \end{pmatrix} = \begin{pmatrix} Y_1^{(0)} \\ Y_2^{(0)} \\ \vdots \\ Y_{s-1}^{(0)} \\ Y_s^{(0)} \end{pmatrix},$$

where

$$\begin{aligned} A_i^{(0)} &= (a_{lm})_{l \in I_i, m \in I_{i-1}} \quad \text{for } i = 2, \ldots, s , \\ B_i^{(0)} &= (a_{lm})_{l \in I_i, m \in I_i} \quad \text{for } i = 1, \ldots, s , \\ C_i^{(0)} &= (a_{lm})_{l \in I_i, m \in I_{i+1}} \quad \text{for } i = 1, \ldots, s-1 \end{aligned}$$

are sub-matrices of A. The index sets are for $i = 1, \ldots, s$

$$I_i = \{ j \in \mathbb{N} \mid (i-1)r < j \leq ir \}.$$

The vectors X_i, $Y_i^{(0)} \in \mathbb{R}^r$ are

$$X_i = (x_l)_{l \in I_i} \text{ and } Y_i^{(0)} = (y_l)_{l \in I_i} \text{ for } i = 1, \ldots, s.$$

The algorithm from above is generalized by applying the described computation steps for elements according to Eq. (7.23) to blocks and using matrix operations instead of operations on single elements. In the first step, three consecutive matrix equations $i-1, i, i+1$ for $i = 3, 4, \ldots, s-2$ are considered:

$$\begin{aligned} A_{i-1}^{(0)} X_{i-2} + B_{i-1}^{(0)} X_{i-1} + C_{i-1}^{(0)} X_i &= Y_{i-1}^{(0)}, \\ A_i^{(0)} X_{i-1} + B_i^{(0)} X_i + C_i^{(0)} X_{i+1} &= Y_i^{(0)}, \\ A_{i+1}^{(0)} X_i + B_{i+1}^{(0)} X_{i+1} + C_{i+1}^{(0)} X_{i+2} &= Y_{i+1}^{(0)}. \end{aligned}$$

Equation $(i-1)$ is used to eliminate subvector X_{i-1} from equation i and equation $(i+1)$ is used to eliminate subvector X_{i+1} from equation i. The algorithm starts with the following initializations:

$$A_1^{(0)} := 0 \in \mathbb{R}^{r \times r}, \; C_s^{(0)} := 0 \in \mathbb{R}^{r \times r}$$

and for $k = 0, \ldots, \lceil \log s \rceil$ and $i \in \mathbb{Z} \setminus \{1, \ldots, s\}$

$$\begin{aligned} A_i^{(k)} = C_i^{(k)} &:= 0 \in \mathbb{R}^{r \times r}, \\ B_i^{(k)} &:= I \in \mathbb{R}^{r \times r}, \\ Y_i^{(k)} &:= 0 \in \mathbb{R}^r. \end{aligned}$$

In step $k = 1, \ldots, \lceil \log s \rceil$ the following sub-matrices

$$\begin{aligned} \alpha_i^{(k)} &:= -A_i^{(k-1)} \left(B_{i-2^{k-1}}^{(k-1)} \right)^{-1}, \\ \beta_i^{(k)} &:= -C_i^{(k-1)} \left(B_{i+2^{k-1}}^{(k-1)} \right)^{-1}, \end{aligned}$$

$$\begin{aligned} A_i^{(k)} &= \alpha_i^{(k)} \cdot A_{i-2^{k-1}}^{(k-1)}, \\ C_i^{(k)} &= \beta_i^{(k)} \cdot C_{i+2^{k-1}}^{(k-1)}, \\ B_i^{(k)} &= \alpha_i^{(k)} C_{i-2^{k-1}}^{(k-1)} + B_i^{(k-1)} + \beta_i^{(k)} A_{i+2^{k-1}}^{(k-1)} \end{aligned} \tag{7.29}$$

and the vector

$$Y_i^{(k)} = \alpha_i^{(k)} Y_{i-2^{k-1}}^{(k-1)} + Y_i^{(k-1)} + \beta_i^{(k)} Y_{i+2^{k-1}}^{(k-1)} \tag{7.30}$$

are computed. The resulting matrix equations are

$$A_i^{(k)} X_{i-2^k} + B_i^{(k)} X_i + C_i^{(k)} X_{i+2^k} = Y_i^{(k)} \tag{7.31}$$

for $i = 1, \ldots, s$. In summary, the method of cyclic reduction for banded matrices comprises the following two phases:

1. Elimination phase: For $k = 1, \ldots, \lceil \log s \rceil$ compute the matrices $A_i^{(k)}$, $B_i^{(k)}$, $C_i^{(k)}$ and the vector $Y_i^{(k)}$ for $i = 2^k, \ldots, s$ with step size 2^k according to Eqs. (7.29) and (7.30).
2. Substitution phase: For $k = \lfloor \log s \rfloor, \ldots, 0$ compute subvector X_i for $i = 2^k, \ldots, s$ with step size 2^{k+1} by solving the linear equation system (7.31), i.e.,

$$B_i^{(k)} X_i = Y_i^{(k)} - A_i^{(k)} X_{i-2^k} - C_i^{(k)} X_{i+2^k} \ .$$

The computation of $\alpha_i^{(k)}$ and $\beta_i^{(k)}$ requires a matrix inversion or the solution of a dense linear equation system with a direct method requiring $O(r^3)$ computations, i.e., the computations increase with the bandwidth cubically. The first step requires the computation of $O(s) = O(n/r)$ sub-matrices; the asymptotic runtime for this step is therefore $O(nr^2)$. The second step solves a total number of $O(s) = O(n/r)$ linear equation systems, also resulting in an asymptotic runtime of $O(nr^2)$.

For the parallel implementation of the cyclic reduction for banded matrices, the parallel method described for tridiagonal systems with its three phases can be used. The main difference is that arithmetic operations in the implementation for tridiagonal systems are replaced by matrix operations in the implementation for banded systems, which increases the amount of computations for each processor. The computational effort for the local operations is now $O(r^3)$. Also, the communication between the processors exchanges larger messages. Instead of single numbers, entire matrices of size $r \times r$ are exchanged so that the message size is $O(r^2)$. Thus, with growing semi-bandwidth r of the banded matrix the time for the computation increases faster than the communication time. For $p \ll s$ an efficient parallel implementation can be expected.

7.2.4 Solving the Discretized Poisson Equation

The cyclic reduction algorithm for banded matrices presented in Sect. 7.2.3 is suitable for the solution of the discretized two-dimensional Poisson equation. As shown in Sect. 7.2.1, this linear equation system has a banded structure with semi-bandwidth N where N is the number of discretization points in the x- or y-dimension of the two-dimensional domain, see Fig. 7.9. The special structure has only four non-zero diagonals and the band has a sparse structure. The use of the Gaussian elimination method would not preserve the sparse banded structure of the matrix, since the forward elimination for eliminating the two lower diagonals leads to fill-ins with non-zero elements between the two upper diagonals. This induces a higher computational effort which is needed for banded matrices with a dense band of semi-bandwidth N. In the following, we consider the method of cyclic reduction for banded matrices, which preserves the sparse banded structure.

The blocks of the discretized Poisson equation $Az = d$ for a representation as blocked tridiagonal matrix are given by Eqs. (7.18) and (7.19). Using the notation for the banded system, we get

$$B_i^{(0)} := \frac{1}{h^2} B \quad \text{for } i = 1, \ldots, N \ ,$$
$$A_i^{(0)} := -\frac{1}{h^2} I \text{ and } C_i^{(0)} := -\frac{1}{h^2} I \quad \text{for } i = 1, \ldots, N \ .$$

The vector $d \in \mathbb{R}^n$ consists of N subvectors $D_j \in \mathbb{R}^N$, i.e.,

$$d = \begin{pmatrix} D_1 \\ \vdots \\ D_N \end{pmatrix} \text{ with } D_j = \begin{pmatrix} d_{(j-1)N+1} \\ \vdots \\ d_{jN} \end{pmatrix} \ .$$

Analogously, the solution vector consists of N subvectors Z_j of length N each, i.e.,

$$z = \begin{pmatrix} Z_1 \\ \vdots \\ Z_N \end{pmatrix} \text{ with } Z_j = \begin{pmatrix} z_{(j-1)N+1} \\ \vdots \\ z_{j \cdot N} \end{pmatrix} \ .$$

The initialization for the cyclic reduction algorithm is given by

$$\begin{aligned} B^{(0)} &:= B \ , \\ D_j^{(0)} &:= D_j \quad \text{for } j = 1, \ldots, N \ , \\ D_j^{(k)} &:= 0 \quad \text{for } k = 0, \ldots, \lceil \log N \rceil, \ j \in \mathbb{Z} \setminus \{1, \ldots, N\} \ , \\ Z_j &:= 0 \quad \text{for } j \in \mathbb{Z} \setminus \{1, \ldots, N\} \ . \end{aligned}$$

In step k of the cyclic reduction, $k = 1, \ldots, \lfloor \log N \rfloor$, the matrices $B^{(k)} \in \mathbb{R}^{N \times N}$ and the vectors $D_j^{(k)} \in \mathbb{R}^N$ for $j = 1, \ldots, N$ are computed according to

$$\begin{aligned} B^{(k)} &= (B^{(k-1)})^2 - 2I \ , \\ D_j^{(k)} &= D_{j-2^{k-1}}^{(k-1)} + B^{(k-1)} D_j^{(k-1)} + D_{j+2^{k-1}}^{(k-1)} \ . \end{aligned} \tag{7.32}$$

For $k = 0, \ldots, \lfloor \log N \rfloor$ Eq. (7.31) has the special form

$$-Z_{j-2^k} + B^{(k)} Z_j - Z_{j+2^k} = D_j^{(k)} \qquad \text{for } j = 1, \ldots, n \ . \tag{7.33}$$

Together Eqs. (7.32) and (7.33) represent the method of cyclic reduction for the discretized Poisson equation, which can be seen by induction. For $k = 0$, Eq. (7.33) is the initial equation system $Az = d$. For $0 < k < \lceil \log N \rceil$ and $j \in \{1, \ldots, N\}$ the three equations

$$\begin{aligned} -Z_{j-2^{k+1}} + B^{(k)} Z_{j-2^k} - Z_j - Z_{j+2^k} &= D^{(k)}_{j-2^k}, \\ -Z_{j-2^k} + B^{(k)} Z_j - Z_{j+2^k} &= D^{(k)}_j, \\ -Z_j + B^{(k)} Z_{j+2^k} - Z_{j+2^{k+1}} &= D^{(k)}_{j+2^k} \end{aligned} \tag{7.34}$$

are considered. The multiplication of Eq. (7.33) with $B^{(k)}$ from the left results in

$$-B^{(k)} Z_{j-2^k} + B^{(k)} B^{(k)} Z_j - B^{(k)} Z_{j+2^k} = B^{(k)} D^{(k)}_j \,. \tag{7.35}$$

Adding Eq. (7.35) with the first part in Eq. (7.34) and the third part in Eq. (7.34) results in

$$-Z_{j-2^{k+1}} - Z_j + B^{(k)} B^{(k)} Z_j - Z_j - Z_{j+2^{k+1}} = D^{(k)}_{j-2^k} + B^{(k)} D^{(k)}_j + D^{(k)}_{j+2^k} \,,$$

which shows that Formula (7.32) for $k+1$ is derived. In summary, the cyclic reduction for the discretized two-dimensional Poisson equation consists of the following two steps:

1. Elimination phase: For $k = 1, \ldots, \lfloor \log N \rfloor$, the matrices $B^{(k)}$ and the vectors $D^{(k)}_j$ are computed for $j = 2^k, \ldots, N$ with step size 2^k according to Eq. (7.32).
2. Substitution phase: For $k = \lfloor \log N \rfloor, \ldots, 0$, the linear equation system

$$B^{(k)} Z_j = D^{(k)}_j + Z_{j-2^k} + Z_{j+2^k}$$

for $j = 2^k, \ldots, N$ with step size 2^{k+1} is solved.

In the first phase, $\lfloor \log N \rfloor$ matrices and $O(N)$ subvectors are computed. The computation of each matrix includes a matrix multiplication with time $O(N^3)$. The computation of a subvector includes a matrix–vector multiplication with complexity $O(N^2)$. Thus, the first phase has a computational complexity of $O(N^3 \log N)$. In the second phase, $O(N)$ linear equation systems are solved. This requires time $O(N^3)$ when the special structure of the matrices $B^{(k)}$ is not exploited. In [61] it is shown how to reduce the time by exploiting this structure. A parallel implementation of the discretized Poisson equation can be done in an analogous way as shown in the previous section.

7.3 Iterative Methods for Linear Systems

In this section, we introduce classical iteration methods for solving linear equation systems, including the Jacobi iteration, the Gauss–Seidel iteration, and the SOR method (successive over-relaxation), and discuss their parallel implementation. Direct methods as presented in the previous sections involve a factorization of the coefficient matrix. This can be impractical for large and sparse matrices, since fill-ins with non-zero elements increase the computational work. For banded

matrices, special methods can be adapted and used as discussed in Sect. 7.2. Another possibility is to use iterative methods as presented in this section.

Iterative methods for solving linear equation systems $Ax = b$ with coefficient matrix $A \in \mathbb{R}^{n \times n}$ and right-hand side $b \in \mathbb{R}^n$ generate a sequence of approximation vectors $\{x^{(k)}\}_{k=1,2,\ldots}$ that converges to the solution $x^* \in \mathbb{R}^n$. The computation of an approximation vector essentially involves a matrix–vector multiplication with the iteration matrix of the specific problem. The matrix A of the linear equation system is used to build this iteration matrix. For the evaluation of an iteration method it is essential how quickly the iteration sequence converges. Basic iteration methods are the Jacobi and the Gauss–Seidel methods, which are also called **relaxation methods** historically, since the computation of a new approximation depends on a combination of the previously computed approximation vectors. Depending on the specific problem to be solved, relaxation methods can be faster than direct solution methods. But still these methods are not fast enough for practical use. A better convergence behavior can be observed for methods like the SOR method, which has a similar computational structure. The practical importance of relaxation methods is their use as preconditioner in combination with solution methods like the conjugate gradient method or the multigrid method. Iterative methods are a good first example to study parallelism as it is typical also for more complex iteration methods. In the following, we describe the relaxation methods according to [23], see also [71, 166]. Parallel implementations are considered in [60, 61, 72, 154].

7.3.1 Standard Iteration Methods

Standard iteration methods for the solution of a linear equation system $Ax = b$ are based on a splitting of the coefficient matrix $A \in \mathbb{R}^{n \times n}$ into

$$A = M - N \quad \text{with } M, N \in \mathbb{R}^{n \times n} ,$$

where M is a non-singular matrix for which the inverse M^{-1} can be computed easily, e.g., a diagonal matrix. For the unknown solution x^* of the equation $Ax = b$ we get

$$Mx^* = Nx^* + b .$$

This equation induces an iteration of the form $Mx^{(k+1)} = Nx^{(k)} + b$, which is usually written as

$$x^{(k+1)} = Cx^{(k)} + d \tag{7.36}$$

with iteration matrix $C := M^{-1}N$ and vector $d := M^{-1}b$. The iteration method is called *convergent* if the sequence $\{x^{(k)}\}_{k=1,2,\ldots}$ converges toward x^* independently of the choice of the start vector $x^{(0)} \in \mathbb{R}^n$, i.e., $\lim_{k\to\infty} x^{(k)} = x^*$ or $\lim_{k\to\infty} \|x^{(k)} - x^*\| = 0$. When a sequence converges the vector x^* is uniquely defined by $x^* = Cx^* + d$. Subtracting this equation from Eq. (7.36) and using induction leads to the

equality $x^{(k)} - x^* = C^k(x^{(0)} - x^*)$, where C^k denotes the matrix resulting from k multiplications of C. Thus, the convergence of Eq. (7.36) is equivalent to

$$\lim_{k\to\infty} C^k = 0 \, .$$

A result from linear algebra shows the relation between the convergence criteria and the spectral radius $\rho(C)$ of the iteration matrix C. (The spectral radius of a matrix is the eigenvalue with the largest absolute value, i.e., $\rho(C) = \max_{\lambda \in EW} |\lambda|$ with $EW = \{\lambda \mid Cv = \lambda v, v \neq 0\}$.) The following properties are equivalent, see [166]:

(1) Iteration (7.36) converges for every $x^{(0)} \in \mathbb{R}^n$.
(2) $\lim_{k\to\infty} C^k = 0$.
(3) $\rho(C) < 1$.

Well-known iteration methods are the Jacobi, the Gauss–Seidel, and the SOR method.

7.3.1.1 Jacobi Iteration

The Jacobi iteration is based on the splitting $A = D - L - R$ of the matrix A with $D, L, R \in \mathbb{R}^{n\times n}$. The matrix D holds the diagonal elements of A, $-L$ holds the elements of the lower triangular of A without the diagonal elements, and $-R$ holds the elements of the upper triangular of A without the diagonal elements. All other elements of D, L, R are zero. The splitting is used for an iteration of the form

$$Dx^{(k+1)} = (L + R)x^{(k)} + b,$$

which leads to the iteration matrix $C_{Ja} := D^{-1}(L + R)$ or

$$C_{Ja} = (c_{ij})_{i,j=1,\ldots,n} \text{ with } c_{ij} = \begin{cases} -a_{ij}/a_{ii} & \text{for } j \neq i \, , \\ 0 & \text{otherwise} \, . \end{cases}$$

The matrix form is used for the convergence proof, not shown here. For the practical computation, the equation written out with all its components is more suitable:

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j=1, j\neq i}^{n} a_{ij} x_j^{(k)} \right) \, , \quad i = 1, \ldots, n \, . \tag{7.37}$$

The computation of one component $x_i^{(k+1)}$, $i \in \{1, \ldots, n\}$, of the $(k + 1)$th approximation requires all components of the kth approximation vector x^k. Considering a sequential computation in the order $x_1^{(k+1)}, \ldots, x_n^{(k+1)}$, it can be observed that the values $x_1^{(k+1)}, \ldots, x_{i-1}^{(k+1)}$ are already known when $x_i^{(k+1)}$ is computed. This information is exploited in the Gauss–Seidel iteration method.

7.3.1.2 Gauss–Seidel Iteration

The Gauss–Seidel iteration is based on the same splitting of the matrix A as the Jacobi iteration, i.e., $A = D - L - R$, but uses the splitting in a different way for an iteration

$$(D - L)x^{(k+1)} = Rx^{(k)} + b \ .$$

Thus, the iteration matrix of the Gauss–Seidel method is $C_{Ga} := (D - L)^{-1}R$; this form is used for numerical properties like convergence proofs, not shown here. The component form for the practical use is

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij} x_j^{(k)} \right) \ , \quad i = 1, \ldots, n \ . \tag{7.38}$$

It can be seen that the components of $x_i^{(k+1)}$, $i \in \{1, \ldots, n\}$, uses the new information $x_1^{(k+1)}, \ldots, x_{i-1}^{(k+1)}$ already determined for that approximation vector. This is useful for a faster convergence in a sequential implementation, but the potential parallelism is now restricted.

7.3.1.3 Convergence Criteria

For the Jacobi and the Gauss–Seidel iteration the following convergence criteria based on the structure of A is often helpful. The Jacobi and the Gauss–Seidel iteration converge if the matrix A is **strongly diagonal dominant**, i.e.,

$$|a_{ii}| > \sum_{j=1, j \neq i}^{n} |a_{ij}| \ , \quad i = 1, \ldots, n \ .$$

When the absolute values of the diagonal elements are large compared to the sum of the absolute values of the other row elements, this often leads to a better convergence. Also, when the iteration methods converge, the Gauss–Seidel iteration often converges faster than the Jacobi iteration, since always the most recently computed vector components are used. Still the convergence is usually not fast enough for practical use. Therefore, an additional relaxation parameter is introduced to speed up the convergence.

7.3.1.4 JOR Method

The JOR method or Jacobi over-relaxation is based on the splitting $A = \frac{1}{\omega} D - L - R - \frac{1-\omega}{\omega} D$ of the matrix A with a **relaxation parameter** $\omega \in \mathbb{R}$. The component form of this modification of the Jacobi method is

$$x_i^{(k+1)} = \frac{\omega}{a_{ii}} \left(b_i - \sum_{j=1, j\neq i}^{n} a_{ij} x_j^{(k)} \right) + (1-\omega) x_i^{(k)} , \quad i = 1, \ldots, n . \tag{7.39}$$

More popular is the modification with a relaxation parameter for the Gauss–Seidel method, the SOR method.

7.3.1.5 SOR Method

The SOR method or (**successive over-relaxation**) is a modification of the Gauss–Seidel iteration that speeds up the convergence of the Gauss–Seidel method by introducing a relaxation parameter $\omega \in \mathbb{R}$. This parameter is used to modify the way in which the combination of the previous approximation $x^{(k)}$ and the components of the current approximation $x_1^{(k+1)}, \ldots, x_{i-1}^{(k+1)}$ are combined in the computation of $x_i^{(k+1)}$. The $(k+1)$th approximation computed according to the Gauss–Seidel iteration (7.38) is now considered as intermediate result $\hat{x}^{(k+1)}$ and the next approximation $x^{(k+1)}$ of the SOR method is computed from both vectors $\hat{x}^{(k+1)}$ and $x^{(k+1)}$ in the following way:

$$\hat{x}_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij} x_j^{(k)} \right) , \quad i = 1, \ldots, n , \tag{7.40}$$

$$x_i^{(k+1)} = x_i^{(k)} + \omega (\hat{x}_i^{(k+1)} - x_i^{(k)}) , \quad i = 1, \ldots, n . \tag{7.41}$$

Substituting Eq. (7.40) into Eq. (7.41) results in the iteration

$$x_i^{(k+1)} = \frac{\omega}{a_{ii}} \left(b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij} x_j^{(k)} \right) + (1-\omega) x_i^{(k)} \tag{7.42}$$

for $i = 1, \ldots, n$. The corresponding splitting of the matrix A is $A = \frac{1}{\omega} D - L - R - \frac{1-\omega}{\omega} D$ and an iteration step in matrix form is

$$(D - wL) x^{(k+1)} = (1-\omega) D x^{(k)} + \omega R x^{(k)} + \omega b .$$

The convergence of the SOR method depends on the properties of A and the value chosen for the relaxation parameter ω. For example the following property holds: If A is symmetric and positive definite and $\omega \in (0, 2)$, then the SOR method converges for every start vector $x^{(0)}$. For more numerical properties see books on numerical linear algebra, e.g., [23, 61, 71, 166].

7.3.1.6 Implementation Using Matrix Operations

The iteration (7.36) computing $x^{(k+1)}$ for a given vector $x^{(k)}$ consists of

- a matrix–vector multiplication of the iteration matrix C with $x^{(k)}$ and
- a vector–vector addition of the result of the multiplication with vector d.

The specific structure of the iteration matrix, i.e., C_{Ja} for the Jacobi iteration and C_{Ga} for the Gauss–Seidel iteration, is exploited. For the Jacobi iteration with $C_{Ja} = D^{-1}(L + R)$ this results in the following computation steps:

- a matrix–vector multiplication of $L + R$ with $x^{(k)}$,
- a vector–vector addition of the result with b, and
- a matrix–vector multiplication with D^{-1} (where D is a diagonal matrix and thus D^{-1} is easy to compute).

A sequential implementation uses Formula (7.37), and the components $x_i^{(k+1)}$, $i = 1, \ldots, n$, are computed one after another. The entire vector $x^{(k)}$ is needed for this computation. For the Gauss–Seidel iteration with $C_{Ga} = (D - L)^{-1}R$ the computation steps are

- a matrix–vector multiplication $Rx^{(k)}$ with upper triangular matrix R,
- a vector–vector addition of the result with b, and
- the solution of a linear system with lower triangular matrix $(D - L)$.

A sequential implementation uses Formula (7.38). Since the most recently computed approximation components are always used for computing a value $x_i^{(k+1)}$, the previous value $x_i^{(k)}$ can be overwritten. The iteration method stops when the current approximation is close enough to the exact solution. Since this solution is unknown, the relative error is used for error control and after each iteration step the convergence is tested according to

$$\|x^{(k+1)} - x^{(k)}\| \leq \varepsilon \|x^{(k+1)}\|, \tag{7.43}$$

where ε is a predefined error value and $\|.\|$ is a vector norm such as $\|x\|_\infty = \max_{i=1,\ldots,n} |x|_i$ or $\|x\|_2 = (\sum_{i=1}^{n} |x_i|^2)^{\frac{1}{2}}$.

7.3.2 Parallel Implementation of the Jacobi Iteration

In the Jacobi iteration (7.37), the computations of the components $x_i^{(k+1)}$, $i = 1, \ldots, n$, of approximation $x^{(k+1)}$ are independent of each other and can be executed in parallel. Thus, each iteration step has a maximum degree of potential parallelism of n and $p = n$ processors can be employed. For a parallel system with distributed memory, the values $x_i^{(k+1)}$ are stored in the individual local memories. Since the computation of one of the components of the next approximation requires all components of the previous approximation, communication has to be performed to create a replicated distribution of $x^{(k)}$. This can be done by a multi-broadcast operation.

When considering the Jacobi iteration built up of matrix and vector operations, a parallel implementation can use the parallel implementations introduced

in Sect. 3.6. The iteration matrix C_{Ja} is not built up explicitly but matrix A is used without its diagonal elements. The parallel computation of the components of $x^{(k+1)}$ corresponds to the parallel implementation of the matrix–vector product using the parallelization with scalar products, see Sect. 3.6. The vector addition can be done after the multi-broadcast operation by each of the processors or before the multi-broadcast operation in a distributed way. When using the parallelization of the linear combination from Sect. 3.6, the vector addition takes place after the accumulation operation. The final broadcast operation is required to provide $x^{(k+1)}$ to all processors also in this case.

Figure 7.13 shows a parallel implementation of the Jacobi iteration using C notation and MPI operations from [135]. For simplicity it is assumed that the matrix size n is a multiple of the number of processors p. The iteration matrix is stored in a row-blockwise way so that each processor owns n/p consecutive rows of matrix A which are stored locally in array `local_A`. The vector b is stored in a corresponding blockwise way. This means that the processor `me`, $0 \leq$ `me` $< p$, stores the rows $\texttt{me} \cdot n/p + 1, \ldots, (\texttt{me} + 1) \cdot n/p$ of A in `local_A` and the corresponding components of b in `local_b`. The iteration uses two local arrays `x_old` and `x_new` for storing the previous and the current approximation vectors. The symbolic constant `GLOB_MAX` is the maximum size of the linear equation system to be solved. The result of the local matrix–vector multiplication is stored in `local_x`; `local_x` is computed according to Eq. (7.37). An `MPI_Allgather()` operation combines the local results so that each processor stores the entire vector `x_new`. The iteration stops when a predefined number `max_it` of iteration steps has been performed or when the difference between `x_old` and `x_new` is smaller than a predefined value `tol`. The function `distance()` implements a maximum norm and the function `output(x_new,global_x)` returns array `global_x` which contains the last approximation vector to be the final result.

7.3.3 Parallel Implementation of the Gauss–Seidel Iteration

The Gauss–Seidel iteration (7.38) exhibits data dependences, since the computation of the component $x_i^{(k+1)}$, $i \in \{1, \ldots, n\}$, uses the components $x_1^{(k+1)}, \ldots, x_{i-1}^{(k+1)}$ of the same approximation and the components of $x^{(k+1)}$ have to be computed one after another. Since for each $i \in \{1, \ldots, n\}$ the computation (7.38) corresponds to a scalar product of the vector

$$(x_1^{(k+1)}, \ldots, x_{i-1}^{(k+1)}, 0, x_{i+1}^{(k)}, \ldots, x_n^{(k)})$$

and the ith row of A, this means that the scalar products have to be computed one after another. Thus, parallelism is only possible within the computation of each single scalar product: Each processor can compute a part of the scalar product, i.e., a local scalar product, and the results are then accumulated. For such an implementation a column-blockwise distribution of matrix A is suitable. Again, we assume that n is a multiple of the number p of processors. The approximation vectors are

```
int Parallel_jacobi(int n, int p, int max_it, float tol)
{
 int i_local, i_global, j, i;
 int n_local, it_num;
 float x_temp1[GLOB_MAX], x_temp2[GLOB_MAX], local_x[GLOB_MAX];
 float *x_old, *x_new, *temp;

 n_local = n/p; /* local blocksize */
 MPI_Allgather(local_b, n_local, MPI_FLOAT, x_temp1, n_local,
              MPI_FLOAT, MPI_COMM_WORLD);
 x_new = x_temp1;
 x_old = x_temp2;
 it_num = 0;
 do {
   it_num ++;
   temp = x_new; x_new = x_old; x_old = temp;
   for (i_local = 0; i_local < n_local; i_local++) {
     i_global = i_local + me * n_local;
     local_x[i_local] = local_b[i_local];
     for (j = 0; j < i_global; j++)
       local_x[i_local] = local_x[i_local] -
                          local_A[i_local][j] * x_old[j];
     for (j = i_global+1 ; j < n; j++)
       local_x[i_local] = local_x[i_local] -
                          local_A[i_local][j] * x_old[j];
     local_x[i_local] = local_x[i_local]/ local_A[i_local][i_global];
   }
   MPI_Allgather(local_x, n_local, MPI_FLOAT, x_new, n_local,
                 MPI_FLOAT, MPI_COMM_WORLD);
 } while ((it_num < max_it) && (distance(x_old,x_new,n) >= tol));
 output(x_new,global_x);
 if (distance(x_old, x_new, n) < tol ) return 1;
 else return 0;
}
```

Fig. 7.13 Program fragment in C notation and with MPI communication operations for a parallel implementation of the Jacobi iteration. The arrays `local_x`, `local_b`, and `local_A` are declared globally. The dimension of `local_A` is `n_local` × `n`. A pointer-oriented storage scheme as shown in Fig. 7.3 is not used here so that the array indices in this implementation differ from the indices in a sequential implementation. The computation of `local_x[i_local]` is performed in two loops with loop index j; the first loop corresponds to the multiplication with array elements in row `i_local` to the left of the main diagonal of A and the second loop corresponds to the multiplication with array elements in row `i_local` to the right of the main diagonal of A. The result is divided by `local_A[i_local][i_global]` which corresponds to the diagonal element of that row in the global matrix A

distributed correspondingly in a blockwise way. Processor P_q, $1 \leq q \leq p$, computes that part of the scalar product for which it owns the columns of A and the components of the approximation vector $x^{(k)}$. This is the computation

$$s_{qi} = \sum_{\substack{j=(q-1)\cdot n/p+1 \\ j<i}}^{q\cdot n/p} a_{ij} x_j^{(k+1)} + \sum_{\substack{j=(q-1)\cdot n/p+1 \\ j>i}}^{q\cdot n/p} a_{ij} x_j^{(k)} \,. \tag{7.44}$$

The intermediate results s_{qi} computed by processors P_q, $q = 1, \ldots, p$, are accumulated by a single-accumulation operation with the addition as reduction operation and the value $x_i^{(k+1)}$ is the result. Since the next approximation vector $x^{(k+1)}$ is expected in a blockwise distribution, the value $x_i^{(k+1)}$ is accumulated at the processor owning the ith component, i.e., $x_i^{(k+1)}$ is accumulated by processor P_q with $q = \lceil i/(n/p) \rceil$. A parallel implementation of the SOR method corresponds to the parallel implementation of the Gauss–Seidel iteration, since both methods differ only in the additional relaxation parameter of the SOR method.

Figure 7.14 shows a program fragment using C notation and MPI operations of a parallel Gauss–Seidel iteration. Since only the most recently computed components of an approximation vector are used in further computations, the component $x_i^{(k)}$ is overwritten by $x_i^{(k+1)}$ immediately after its computation. Therefore, only one array `x` is needed in the program. Again, an array `local_A` stores the local part of matrix A which is a block of columns in this case; `n_local` is the size of the block. The `for` loop with loop index i computes the scalar products sequentially; within the loop body the parallel computation of the inner product is performed according to Formula (7.44). An MPI reduction operation computes the components at differing processors `root` which finalizes the computation.

7.3.4 Gauss–Seidel Iteration for Sparse Systems

The potential parallelism for the Gauss–Seidel iteration or the SOR method is limited because of data dependences so that a parallel implementation is only reasonable for very large equation systems. Each data dependency in Formula (7.38) is caused by a coefficient (a_{ij}) of matrix A, since the computation of $x_i^{(k+1)}$ depends on the value $x_j^{(k+1)}$, $j < i$, when $(a_{ij}) \neq 0$. Thus, for a linear equation system $Ax = b$ with sparse matrix $A = (a_{ij})_{i,j=1,\ldots,n}$ there is a larger degree of parallelism caused by less data dependences. If $a_{ij} = 0$, then the computation of $x_i^{(k+1)}$ does not depend on $x_j^{(k+1)}$, $j < i$. For a sparse matrix with many zero elements the computation of $x_i^{(k+1)}$ only needs a few $x_j^{(k+1)}$, $j < i$. This can be exploited to compute components of the $(k+1)$th approximation $x^{(k+1)}$ in parallel.

In the following, we consider sparse matrices with a banded structure like the discretized Poisson equation, see Eq. (7.13) in Sect. 7.2.1. The computation of $x_i^{(k+1)}$ uses the elements in the ith row of A, see Fig. 7.9, which has non-zero elements a_{ij}

```
n_local = n/p;
do {
   delta_x = 0.0;
   for (i = 0; i < n; i++) {
      s_k = 0.0;
      for (j = 0; j < n_local; j++)
         if (j + me * n_local != i)
           s_k = s_k + local_A[i][j] * x[j];
      root = i/n_local;
      i_local = i % n_local;
      MPI_Reduce(&s_k, &x[i_local], 1, MPI_FLOAT, MPI_SUM, root,
                 MPI_COMM_WORLD);
      if (me == root) {
        x_new = (b[i_local] - x[i_local]) / local_A[i][i_local];
        delta_x = max(delta_x, abs(x[i_local] - x_new));
        x[i_local] = x_new;
      }
   }
   MPI_Allreduce(&delta_x, &global_delta, 1, MPI_FLOAT,
                 MPI_MAX, MPI_COMM_WORLD);
} while(global_delta > tol);
```

Fig. 7.14 Program fragment in C notation and using MPI operations for a parallel Gauss–Seidel iteration for a dense linear equation system. The components of the approximations are computed one after another according to Formula (7.38), but each of these computations is done in parallel by all processors. The matrix is stored in a column-blockwise way in the local arrays `local_A`. The vectors x and b are also distributed blockwise. Each processor computes the local error and stores it in `delta_x`. An `MPI_Allreduce()` operation computes the global error `global_delta` from these values so that each processor can perform the convergence test `global_delta > tol`

for $j = i - \sqrt{n}, i-1, i, i+1, i+\sqrt{n}$. Formula (7.38) of the Gauss–Seidel iteration for the discretized Poisson equation has the specific form

$$x_i^{(k+1)} = \frac{1}{a_{ii}}\Big(b_i - a_{i,i-\sqrt{n}} \cdot x_{i-\sqrt{n}}^{(k+1)} - a_{i,i-1} \cdot x_{i-1}^{(k+1)} - a_{i,i+1} \cdot x_{i+1}^{(k)} - a_{i,i+\sqrt{n}} \cdot x_{i+\sqrt{n}}^{(k)}\Big), \quad i = 1, \ldots, n\,. \tag{7.45}$$

Thus, the two values $x_{i-\sqrt{n}}^{(k+1)}$ and $x_{i-1}^{(k+1)}$ have to be computed before the computation of $x_i^{(k+1)}$. The dependences of the values $x_i^{(k+1)}$, $i = 1, \ldots, n$, on $x_j^{(k+1)}$, $j < i$, are illustrated in Fig. 7.15(a) for the corresponding mesh of the discretized physical domain. The computation of $x_i^{(k+1)}$ corresponds to the mesh point i, see also Sect. 7.2.1. In this mesh, the computation of $x_i^{(k+1)}$ depends on all computations for mesh points which are located in the upper left part of the mesh. On the other hand,

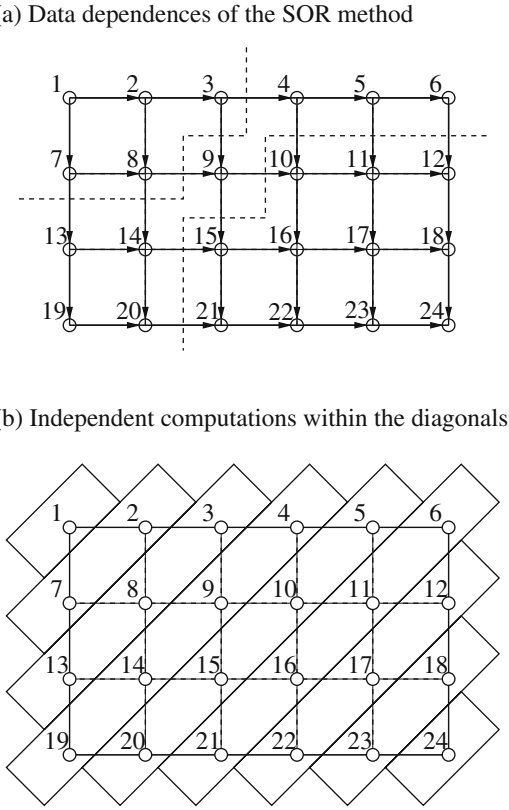


Fig. 7.15 Data dependence of the Gauss–Seidel and the SOR method for a rectangular mesh of size 6×4 in the x–y plane. (**a**) The data dependences between the computations of components are depicted as *arrows* between nodes in the mesh. As an example, for mesh point 9 the set of nodes which have to be computed before point 9 and the set of nodes which depend on mesh point 9 are shown. (**b**) The data dependences lead to areas of independent computations; these are the diagonals of the mesh from the *upper right* to the *lower left*. The computations for mesh points within the same diagonal can be computed in parallel. The length of the diagonals is the degree of potential parallelism which can be exploited

computations for mesh points $j > i$ which are located to the right or below mesh point i need value $x_i^{(k+1)}$ and have to wait for its computation.

The data dependences between computations associated with mesh points are depicted in the mesh by arrows between the mesh points. It can be observed that the mesh points in each diagonal from left to right are independent of each other; these independent mesh points are shown in Fig. 7.15(b). For a square mesh of size $\sqrt{n} \times \sqrt{n}$ with the same number of mesh points in each dimension, there are at most $\sqrt{n}$ independent computations in a single diagonal and at most $p = \sqrt{n}$ processors can be employed.

A parallel implementation can exploit the potential parallelism in a loop structure with an outer sequential loop and an inner parallel loop. The outer sequential loop visits the diagonals one after another from the upper left corner to the lower right corner. The inner loop exploits the parallelism within each diagonal of the mesh. The number of diagonals is $2\sqrt{n} - 1$ consisting of $\sqrt{n}$ diagonals in the upper left triangular mesh and $\sqrt{n} - 1$ in the lower triangular mesh. The first $\sqrt{n}$ diagonals $l = 1, \ldots, \sqrt{n}$ contain l mesh points i with

$$i = l + j \cdot (\sqrt{n} - 1) \quad \text{for} \quad 0 \leq j < l \; .$$

The last $\sqrt{n}-1$ diagonals $l = 2, \ldots, \sqrt{n}$ contain $\sqrt{n}-l+1$ mesh points i with

$$i = l \cdot \sqrt{n} + j \cdot (\sqrt{n}-1) \quad \text{for} \quad 0 \le j \le \sqrt{n}-l \; .$$

For an implementation on a distributed memory machine, a distribution of the approximation vector x, the right-hand side b, and the coefficient matrix A is needed. The elements a_{ij} of matrix A are distributed in such a way that the coefficients for the computation of $x_i^{(k+1)}$ according to Formula (7.45) are locally available. Because the computations are closely related to the mesh, the data distribution is chosen for the mesh and not the matrix form.

The program fragment with C notation in Fig. 7.16 shows a parallel SPMD implementation. The data distribution is chosen such that the data associated with

```
sqn = sqrt(n);
do {
   for (l = 1; l <= sqn; l++) {
      for (j = me; j < l; j+=p) {
         i = l + j * (sqn-1) - 1; /* start numbering with 0 */
         x[i] = 0;
         if (i-sqn >= 0) x[i] = x[i] - a[i][i-sqn] * x[i-sqn];
         if (i > 0) x[i] = x[i] - a[i][i-1] * x[i-1];
         if (i+1 < n) x[i] = x[i] - a[i][i+1] * x[i+1];
         if (i+sqn < n) x[i] = x[i] - a[i][i+sqn] * x[i+sqn];
         x[i] = (x[i] + b[i]) / a[i][i];
      }
      collect_elements(x,l);
   }
   for (l = 2; l <= sqn; l++) {
      for (j = me -l +1; j <= sqn -l; j+=p) {
         if (j >= 0) {
           i = l * sqn + j * (sqn-1) - 1;
           x[i] = 0;
           if (i-sqn >= 0) x[i] = x[i] - a[i][i-sqn] * x[i-sqn];
           if (i > 0) x[i] = x[i] - a[i][i-1] * x[i-1];
           if (i+1 < n) x[i] = x[i] - a[i][i+1] * x[i+1];
           if (i+sqn < n) x[i] = x[i] - a[i][i+sqn] * x[i+sqn];
           x[i] = (x[i] + b[i]) / a[i][i];
         }
      }
      collect_elements(x,l);
   }
} while(convergence_test() < tol);
```

Fig. 7.16 Program fragment of the parallel Gauss–Seidel iteration for a linear equation system with the banded matrix from the discretized Poisson equation. The computational structure uses the diagonals of the corresponding discretization mesh, see Fig. 7.15

mesh points in the same mesh row are stored in the same processor. A row-cyclic distribution of the mesh data is used. The program has two loop nests: The first loop nest treats the upper diagonals and the second loop nest treats the last diagonals. In the inner loops, the processor with name `me` computes the mesh points which are assigned to it due to the row-cyclic distribution of mesh points. The function `collect_elements()` sends the data computed to the neighboring processor, which needs them for the computation of the next diagonal. The function `convergence_test()`, not expressed explicitly in this program, can be implemented similarly as in the program in Fig. 7.14 using the maximum norm for $x^{(k+1)} - x^{(k)}$.

The program fragment in Fig. 7.16 uses two-dimensional indices for accessing array elements of array `a`. For a large sparse matrix, a storage scheme for sparse matrices would be used in practice. Also, for a problem such as the discretized Poisson equation where the coefficients are known it is suitable to code them directly as constants into the program. This saves expensive array accesses but the code is less flexible to solve other linear equation systems.

For an implementation on a shared memory machine, the inner loop is performed in parallel by $p = \sqrt{n}$ processors in an SPMD pattern. No data distribution is needed but the same distribution of work to processors is assigned. Also, no communication is needed to send data to neighboring processors. However, a barrier synchronization is used instead to make sure that the data of the previous diagonal are available for the next one.

A further increase of the potential parallelism for solving sparse linear equation systems can be achieved by the method described in the next section.

7.3.5 Red–Black Ordering

The potential parallelism of the Gauss–Seidel iteration or the successive overrelaxation for sparse systems resulting from discretization problems can be increased by an alternative ordering of the unknowns and equations. The goal of the reordering is to get an equivalent equation system in which more independent computations exist and, thus, a higher potential parallelism results. The most frequently used reordering technique is the **red–black ordering**. The two-dimensional mesh is regarded as a checkerboard where the points of the mesh represent the squares of the checkerboard and get corresponding colors. The point (i, j) in the mesh is colored according to the value of $i + j$: If $i + j$ is even, then the mesh point is red, and if $i + j$ is odd, then the mesh point is black.

The points in the grid now form two sets of points. Both sets are numbered separately in a rowwise way from left to right. First the red points are numbered by $1, \ldots, n_R$ where n_R is the number of red points. Then, the black points are numbered by $n_R + 1, \ldots, n_R + n_B$ where n_B is the number of black points and $n = n_R + n_B$. The unknowns associated with the mesh points get the same numbers as the mesh points: There are n_R unknowns associated with the red points denoted as $\hat{x}_1, \ldots, \hat{x}_{n_R}$ and n_B unknowns associated with the black points denoted as $\hat{x}_{n_R+1}, \ldots, \hat{x}_{n_R+n_B}$. (The notation $\hat{x}$ is used to distinguish the new ordering from the original ordering

of the unknowns x. The unknowns are the same as before but their positions in the system differ.) Figure 7.17 shows a mesh of size 6×4 in its original rowwise numbering in part (a) and a red–black ordering with the new numbering in part (b).

In a linear equation system using red–black ordering, the equations of red unknowns are arranged before the equations with the black unknown. The equation system $\hat{A}\hat{x} = \hat{b}$ for the discretized Poisson equation has the form

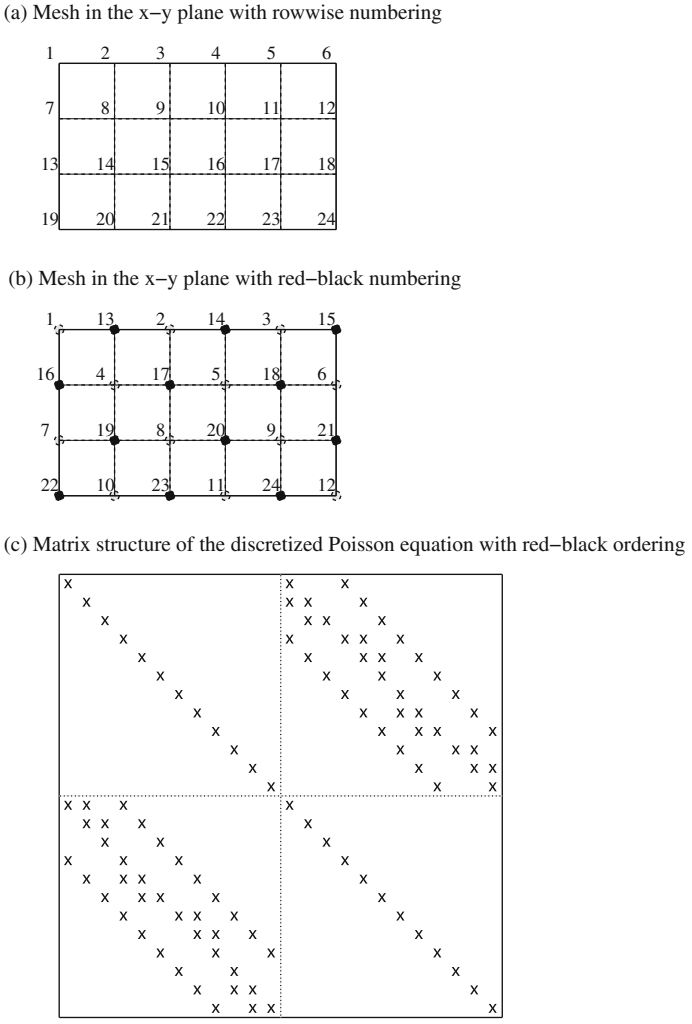


Fig. 7.17 Rectangular mesh in the x–y plane of size 6×4 with (**a**) rowwise numbering, (**b**) red–black numbering, and (**c**) the matrix of the corresponding linear equation system of the five-point formula with red–black numbering

$$\hat{A} \cdot \hat{x} = \begin{pmatrix} D_R & F \\ E & D_B \end{pmatrix} \cdot \begin{pmatrix} \hat{x}_R \\ \hat{x}_B \end{pmatrix} = \begin{pmatrix} \hat{b}_1 \\ \hat{b}_2 \end{pmatrix}, \tag{7.46}$$

where $\hat{x}_R$ denotes the subvector of size n_R of the first (red) unknowns and $\hat{x}_B$ denotes the subvector of size n_B of the last (black) unknowns. The right-hand side b of the original equation system is reordered accordingly and has subvector $\hat{b}_1$ for the first n_R equations and subvector $\hat{b}_2$ for the last n_B equations. The matrix $\hat{A}$ consists of four blocks $D_R \in \mathbb{R}^{n_R \times n_R}$, $D_B \in \mathbb{R}^{n_B \times n_B}$, $E \in \mathbb{R}^{n_B \times n_R}$, and $F \in \mathbb{R}^{n_R \times n_B}$. The submatrices D_R and D_B are diagonal matrices and the submatrices E and F are sparse banded matrices. The structure of the original matrix of the discretized Poisson equation in Fig. 7.9 in Sect. 7.2.1 is thus transformed into a matrix $\hat{A}$ with the structure shown in Fig. 7.17(c).

The diagonal form of the matrices D_R and D_B shows that a red unknown $\hat{x}_i$, $i \in \{1, \ldots, n_R\}$, does not depend on the other red unknowns and a black unknown $\hat{x}_j$, $j \in \{n_R + 1, \ldots, n_R + n_B\}$, does not depend on the other black unknowns. The matrices E and F specify the dependences between red and black unknowns. The row i of matrix F specifies the dependences of the red unknowns $\hat{x}_i$ ($i < n_R$) on the black unknowns $\hat{x}_j$, $j = n_R + 1, \ldots, n_R + n_B$. Analogously, a row of matrix E specifies the dependences of the corresponding black unknowns on the red unknowns.

The transformation of the original linear equation system $Ax = b$ into the equivalent system $\hat{A}\hat{x} = \hat{b}$ can be expressed by a permutation $\pi : \{1, \ldots, n\} \to \{1, \ldots, n\}$. The permutation maps a node $i \in \{1, \ldots, n\}$ of the rowwise numbering onto the number $\pi(i)$ of the red–black numbering in the following way:

$$x_i = \hat{x}_{\pi(i)}, \quad b_i = \hat{b}_{\pi(i)}, \quad i = 1, \ldots, n \text{ or } x = P\hat{x} \text{ and } b = P\hat{b}$$

with a permutation matrix $P = (P_{ij})_{i,j=1,\ldots,n}$, $P_{ij} = \begin{cases} 1 & \text{if} \quad j = \pi(i) \\ 0 & \text{otherwise} \end{cases}$. For the matrices A and $\hat{A}$ the equation $\hat{A} = P^T A P$ holds. Since for a permutation matrix the inverse is equal to the transposed matrix, i.e., $P^T = P^{-1}$, this leads to $\hat{A}\hat{x} = P^T A P P^T x = P^T b = \hat{b}$. The easiest way to exploit the red–black ordering is to use an iterative solution method as discussed earlier in this section.

7.3.5.1 Gauss–Seidel Iteration for Red–Black Systems

The solution of the linear equation system (7.46) with the Gauss–Seidel iteration is based on a splitting of the matrix $\hat{A}$ of the form $\hat{A} = \hat{D} - \hat{L} - \hat{U}$, $\hat{D}, \hat{L}, \hat{U} \in \mathbb{R}^{n \times n}$,

$$\hat{D} = \begin{pmatrix} D_R & 0 \\ 0 & D_B \end{pmatrix}, \quad \hat{L} = \begin{pmatrix} 0 & 0 \\ -E & 0 \end{pmatrix}, \quad \hat{U} = \begin{pmatrix} 0 & -F \\ 0 & 0 \end{pmatrix},$$

with a diagonal matrix $\hat{D}$, a lower triangular matrix $\hat{L}$, and an upper triangular matrix $\hat{U}$. The matrix 0 is a matrix in which all entries are 0. With this notation, iteration step k of the Gauss–Seidel method is given by

$$\begin{pmatrix} D_R & 0 \\ E & D_B \end{pmatrix} \cdot \begin{pmatrix} x_R^{(k+1)} \\ x_B^{(k+1)} \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \end{pmatrix} - \begin{pmatrix} 0 & F \\ 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} x_R^{(k)} \\ x_B^{(k)} \end{pmatrix} \tag{7.47}$$

for $k = 1, 2, \ldots$. According to equation system (7.46), the iteration vector is split into two subvectors $x_R^{(k+1)}$ and $x_B^{(k+1)}$ for the red and the black unknowns, respectively. (To simplify the notation, we use x_R instead of $\hat{x}_R$ in the following discussion of the red–black ordering.)

The linear equation system (7.47) can be written in vector notation for vectors $x_R^{(k+1)}$ and $x_B^{(k+1)}$ in the form

$$D_R \cdot x_R^{(k+1)} = b_1 - F \cdot x_B^{(k)} \quad \text{for } k = 1, 2, \ldots , \tag{7.48}$$

$$D_B \cdot x_B^{(k+1)} = b_2 - E \cdot x_R^{(k+1)} \quad \text{for } k = 1, 2, \ldots , \tag{7.49}$$

in which the decoupling of the red subvector $x_R^{(k+1)}$ and the black subvector $x_B^{(k+1)}$ becomes obvious: In Eq. (7.48) the new red iteration vector $x_R^{(k+1)}$ depends only on the previous black iteration vector $x_B^{(k)}$ and in Eq. (7.49) the new black iteration vector $x_B^{(k+1)}$ depends only on the red iteration vector $x_R^{(k+1)}$ computed before in the same iteration step. There is no additional dependence. Thus, the potential degree of parallelism in Eq. (7.48) or (7.49) is similar to the potential parallelism in the Jacobi iteration. In each iteration step k, the components of $x_R^{(k+1)}$ according to Eq. (7.48) can be computed independently, since the vector $x_B^{(k)}$ is known, which leads to a potential parallelism with $p = n_R$ processors. Afterwards, the vector $x_R^{(k+1)}$ is known and the components of the vector $x_B^{(k+1)}$ can be computed independently according to Eq. (7.49), leading to a potential parallelism of $p = n_R$ processors.

For a parallel implementation, we consider the Gauss–Seidel iteration of the red–black ordering (7.48) and (7.49) written out in a component-based form:

$$\left(x_R^{(k+1)}\right)_i = \tfrac{1}{\hat{a}_{ii}} \left(\hat{b}_i - \sum_{j \in N(i)} \hat{a}_{ij} \cdot (x_B^{(k)})_j \right), \quad i = 1, \ldots, n_R ,$$

$$\left(x_B^{(k+1)}\right)_i = \tfrac{1}{\hat{a}_{i+n_R, i+n_R}} \left(\hat{b}_{i+n_R} - \sum_{j \in N(i)} \hat{a}_{i+n_R, j} \cdot (x_R^{(k+1)})_j \right), \; i = 1, \ldots, n_B .$$

The set $N(i)$ denotes the set of adjacent mesh points for mesh point i. According to the red–black ordering, the set $N(i)$ contains only black mesh points for a red point i and vice versa. An implementation on a shared memory machine can employ at most $p = n_R$ or $p = n_B$ processors. There are no access conflicts for the parallel computation of $x_R^{(k)}$ or $x_B^{(k)}$ but a barrier synchronization is needed between the two computation phases. The implementation on a distributed memory machine requires a distribution of computation and data. As discussed before for the parallel SOR method, it is useful to distribute the data according to the mesh structure

such that the processor P_q to which the mesh point i is assigned is responsible for the computation or update of the corresponding component of the approximation vector. In a row-oriented distribution of a squared mesh with $\sqrt{n} \times \sqrt{n} = n$ mesh points to p processors, $\sqrt{n}/p$ rows of the mesh are assigned to each processor P_q, $q \in \{1, \ldots, p\}$. In the red–black coloring this means that each processor owns $\frac{1}{2}\frac{n}{p}$ red and $\frac{1}{2}\frac{n}{p}$ black mesh points. (For simplicity we assume that $\sqrt{n}$ is a multiple of p.) Thus, the mesh points

$$(q-1) \cdot \frac{n_R}{p} + 1, \ldots, q \cdot \frac{n_R}{p} \quad \text{for } q = 1, \ldots, p \quad \text{and}$$
$$(q-1) \cdot \frac{n_B}{p} + 1 + n_R, \ldots, q \cdot \frac{n_B}{p} + n_R \quad \text{for} \quad q = 1, \ldots, p$$

are assigned to processor P_q. Figure 7.18 shows an SPMD program implementing the Gauss–Seidel iteration with red–black ordering. The coefficient matrix A is stored according to the pointer-based scheme introduced earlier in Fig. 7.3. After the computation of the red components `xr`, a function `collect_elements(xr)` distributes the red vector to all other processors for the next computation. Analogously, the black vector `xb` is distributed after its computation. The function `collect_elements()` can be implemented by a multi-broadcast operation.

```
local_nr = nr/p; local_nb = nb/p;
do {
   mestartr = me * local_nr;
   for  (i= mestartr; i < mestartr + local_nr; i++) {
       xr[i] = 0;
       for  (j ∈ N(i))
           xr[i] = xr[i] - a[i][j] * xb[j];
       xr[i] = (xr[i]+b[i]) / a[i][i] ;
   }
   collect_elements(xr);
   mestartb = me * local_nb + nr;
   for (i= mestartb; i < mestartb + local_nb; i++) {
       xb[i] = 0;
       for (j ∈ N(i))
           xb[i] = xb[i] - a[i+nr][j] * xr[j];
       xb[i]= (xb[i] + b[i+nr]) / a[i+nr][i+nr];
   }
   collect_elements(xb);
} while (convergence_test());
```

Fig. 7.18 Program fragment for the parallel implementation of the Gauss–Seidel method with the red–black ordering. The arrays `xr` and `xb` denote the unknowns corresponding to the red or black mesh points. The processor number of the executing processor is stored in `me`

7.3.5.2 SOR Method for Red–Black Systems

An SOR method for the linear equation system (7.46) with relaxation parameter ω can be derived from the Gauss–Seidel computation (7.48) and (7.49) by using the combination of the new and the old approximation vectors as introduced in Formula (7.41). One step of the SOR method has then the form

$$\begin{aligned}
\tilde{x}_R^{(k+1)} &= D_R^{-1} \cdot b_1 - D_R^{-1} \cdot F \cdot x_B^{(k)} \,, \\
\tilde{x}_B^{(k+1)} &= D_B^{-1} \cdot b_2 - D_B^{-1} \cdot E \cdot x_R^{(k+1)} \,, \\
x_R^{(k+1)} &= x_R^{(k)} + \omega \left(\tilde{x}_R^{(k+1)} - x_R^{(k)} \right) \,, \\
x_B^{(k+1)} &= x_B^{(k)} + \omega \left(\tilde{x}_B^{(k+1)} - x_B^{(k)} \right) \,, \quad k = 1, 2, \ldots \,.
\end{aligned} \tag{7.50}$$

The corresponding splitting of matrix $\hat{A}$ is $\hat{A} = \frac{1}{\omega}\hat{D} - \hat{L} - \hat{U} - \frac{1-\omega}{\omega}\hat{D}$ with the matrices $\hat{D}, \hat{L}, \hat{U}$ introduced above. This can be written using block matrices:

$$\begin{aligned}
&\begin{pmatrix} D_R & 0 \\ \omega E & D_B \end{pmatrix} \cdot \begin{pmatrix} x_R^{(k+1)} \\ x_B^{(k+1)} \end{pmatrix} \\
&= (1-\omega) \begin{pmatrix} D_R & 0 \\ 0 & D_B \end{pmatrix} \cdot \begin{pmatrix} x_R^{(k)} \\ x_B^{(k)} \end{pmatrix} - \omega \begin{pmatrix} 0 & F \\ 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} x_R^{(k)} \\ x_B^{(k)} \end{pmatrix} + \omega \begin{pmatrix} b_1 \\ b_2 \end{pmatrix}.
\end{aligned} \tag{7.51}$$

For a parallel implementation the component form of this system is used. On the other hand, for the convergence results the matrix form and the iteration matrix have to be considered. Since the iteration matrix of the SOR method for a given linear equation system $Ax = b$ with a certain order of the equations and the iteration matrix of the SOR method for the red–black system $\hat{A}\hat{x} = \hat{b}$ are different, convergence results cannot be transferred. The iteration matrix of the SOR method with red–black ordering is

$$\hat{S}_\omega = \left(\frac{1}{\omega}\hat{D} - \hat{L} \right)^{-1} \left(\frac{1-\omega}{\omega}\hat{D} + \hat{U} \right).$$

For a convergence of the method it has to be shown that $\rho(\hat{S}_\omega) < 1$ for the spectral radius of $\hat{S}_\omega$ and $\omega \in \mathbb{R}$. In general, the convergence cannot be derived from the convergence of the SOR method for the original system, since $P^T S_\omega P$ is not identical to $\hat{S}_\omega$, although $P^T A P = \hat{A}$ holds. However, for the specific case of the model problem, i.e., the discretized Poisson equation, the convergence can be shown. Using the equality $P^T A P = \hat{A}$, it follows that $\hat{A}$ is symmetric and positive definite and, thus, the method converges for the model problem, see [61].

Figure 7.19 shows a parallel SPMD implementation of the SOR method for the red–black ordered discretized Poisson equation. The elements of the coefficient matrix are coded as constants. The unknowns are stored in a two-dimensional structure corresponding to the two-dimensional mesh and not as vector so that

```
do {
   for ((i,j) ∈ myregion) {
      if (is_red(i,j))
        x[i][j] = omega/4 * (h*h*f[i][j] + x[i][j-1] + x[i][j+1]
                  + x[i-1][j] + x[i+1][j]) + (1- omega) * x[i][j];
   }
   exchange_red_borders(x);
   for ((i,j) ∈ myregion) {
      if (is_black(i,j))
        x[i][j] = omega/4 * (h*h*f[i][j] + x[i][j-1] + x[i][j+1]
                  + x[i-1][j] + x[i+1][j]) + (1- omega) * x[i][j];
   }
   exchange_black_borders(x);
} while (convergence_test());
```

Fig. 7.19 Program fragment of a parallel SOR method for a red–black ordered discretized Poisson equation

unknowns appear as `x[i][j]` in the program. The mesh points and the corresponding computations are distributed among the processors; the mesh points belonging to a specific processor are stored in `myregion`. The color red or black of a mesh point (i, j) is an additional attribute which can be retrieved by the functions `is_red()` and `is_black()`. The value `f[i][j]` denotes the discretized right-hand side of the Poisson equation as described earlier, see Eq. (7.15). The functions `exchange_red_borders()` and `exchange_black_borders()` exchange the red or black data of the red or black mesh points between neighboring processors.

7.4 Conjugate Gradient Method

The conjugate gradient method or CG method is a solution method for linear equation systems $Ax = b$ with symmetric and positive definite matrix $A \in \mathbb{R}^{n \times n}$, which has been introduced in [86]. (A is symmetric if $a_{ij} = a_{ji}$ and positive definite if $x^T Ax > 0$ for all $x \in \mathbb{R}^n$ with $x \neq 0$.) The CG method builds up a solution $x^* \in \mathbb{R}^n$ in at most n steps in the absence of roundoff errors. Considering roundoff errors more than n steps may be needed to get a good approximation of the exact solution x^*. For sparse matrices a good approximation of the solution can be achieved in less than n steps, also with roundoff errors [150]. In practice, the CG method is often used as preconditioned CG method which combines a CG method with a preconditioner [154]. Parallel implementations are discussed in [72, 133, 134, 154]; [155] gives an overview. In this section, we present the basic CG method and parallel implementations according to [23, 71, 166].

7.4.1 Sequential CG Method

The CG method exploits an equivalence between the solution of a linear equation system and the minimization of a function.

More precisely, the solution x^* of the linear equation system $Ax = b$, $A \in \mathbb{R}^{n \times n}$, $b \in \mathbb{R}^n$, is the minimum of the function $\Phi : M \subset \mathbb{R}^n \to \mathbb{R}$ with

$$\Phi(x) = \frac{1}{2} x^T A x - b^T x \ , \tag{7.52}$$

if the matrix A is symmetric and positive definite. A simple method to determine the minimum of the function Φ is the method of the steepest gradient [71] which uses the negative gradient. For a given point $x_c \in \mathbb{R}^n$ the function decreases most rapidly in the direction of the negative gradient. The method computes the following two steps:

(a) Computation of the negative gradient $d_c \in \mathbb{R}^n$ at point x_c:

$$d_c = - \text{ grad } \Phi(x_c) = - \left(\frac{\partial}{\partial x_1} \Phi(x_c), \ldots , \frac{\partial}{\partial x_n} \Phi(x_c) \right) = b - A x_c .$$

(b) Determination of the minimum of Φ in the set

$$\{ x_c + t d_c \mid t \geq 0 \} \cap M \ ,$$

which forms a line in $\mathbb{R}^n$ (line search). This is done by inserting $x_c + t d_c$ into Formula (7.52). Using $d_c = b - A x_c$ and the symmetry of matrix A we get

$$\Phi(x_c + t d_c) = \Phi(x_c) - t d_c^T d_c + \frac{1}{2} t^2 d_c^T A d_c \ . \tag{7.53}$$

The minimum of this function with respect to $t \in \mathbb{R}$ can be determined using the derivative of this function with respect to t. The minimum is

$$t_c = \frac{d_c^T d_c}{d_c^T A d_c} . \tag{7.54}$$

The steps (a) and (b) of the method of the steepest gradient are used to create a sequence of vectors $x_k, k = 0, 1, 2, \ldots$, with $x_0 \in \mathbb{R}^n$ and $x_{k+1} = x_k + t_k d_k$. The sequence $(\Phi(x_k))_{k=0,1,2,\ldots}$ is monotonically decreasing which can be seen by inserting Formula (7.54) into Formula (7.53). The sequence converges toward the minimum but the convergence might be slow [71].

The CG method uses a technique to determine the minimum which exploits orthogonal search directions in the sense of **conjugate** or ***A*-orthogonal** vectors d_k. For a given matrix A, which is symmetric and non-singular, two vectors $x, y \in \mathbb{R}^n$ are called conjugate or A-orthogonal, if $x^T A y = 0$. If A is positive definite, k

pairwise conjugate vectors $d_0, \ldots, d_{k-1}$ (with $d_i \neq 0, i = 0, \ldots, k-1$ and $k \leq n$) are linearly independent [23]. Thus, the unknown solution vector x^* of $Ax = b$ can be represented as a linear combination of the conjugate vectors $d_0, \ldots, d_{n-1}$, i.e.,

$$x^* = \sum_{k=0}^{n-1} t_k d_k. \tag{7.55}$$

Since the vectors are orthogonal, $d_k^T A x^* = \sum_{l=0}^{n-1} d_k^T A t_l d_l = t_k d_k^T A d_k$. This leads to

$$t_k = \frac{d_k A x^*}{d_k^T A d_k} = \frac{d_k^T b}{d_k^T A d_k}$$

for the coefficients t_k. Thus, when the orthogonal vectors are known, the values t_k, $k = 0, \ldots, n-1$, can be computed from the right-hand side b.

The algorithm for the CG method uses a representation

$$x^* = x_0 + \sum_{i=0}^{n-1} \alpha_i d_i \tag{7.56}$$

of the unknown solution vector x^* as a sum of a starting vector x_0 and a term $\sum_{i=0}^{n-1} \alpha_i d_i$ to be computed. The second term is computed recursively by

Select $x_0 \in \mathbb{R}^n$
Set $d_0 = -g_0 = b - Ax_0$
While $(||g_k|| > \varepsilon)$ compute for $k = 0, 1, 2, \ldots$

(1) $w_k = Ad_k$

(2) $\alpha_k = \dfrac{g_k^T g_k}{d_k^T w_k}$

(3) $x_{k+1} = x_k + \alpha_k d_k$

(4) $g_{k+1} = g_k + \alpha_k w_k$

(5) $\beta_k = \dfrac{g_{k+1}^T g_{k+1}}{g_k^T g_k}$

(6) $d_{k+1} = -g_{k+1} + \beta_k d_k$

Fig. 7.20 Algorithm of the CG method. (1) and (2) compute the values α_k according to Eq. (7.58). The vector w_k is used for the intermediate result Ad_k. (3) is the computation given in Formula (7.57). (4) computes g_{k+1} for the next iteration step according to Formula (7.58) in a recursive way: $g_{k+1} = Ax_{k+1} - b = A(x_k + \alpha_k d_k) - b = g_k + A\alpha_k d_k$. This vector g_{k+1} represents the error between the approximation x_k and the exact solution. (5) and (6) compute the next vector d_{k+1} of the set of conjugate gradients

$$x_{k+1} = x_k + \alpha_k d_k, \quad k = 1, 2, \ldots, \quad \text{with} \tag{7.57}$$

$$\alpha_k = \frac{-g_k^T d_k}{d_k^T A d_k} \quad \text{and} \quad g_k = Ax_k - b\ . \tag{7.58}$$

Formulas (7.57) and (7.58) determine x^* according to Eq. (7.56) by computing α_i and adding $\alpha_i d_i$ in each step, $i = 1, 2, \ldots$. Thus, the solution is computed after at most n steps. If not all directions d_k are needed for x^*, less than n steps are required.

Algorithms implementing the CG method do not choose the conjugate vectors $d_0, \ldots, d_{n-1}$ before computing the vectors x_0,..., x_{n-1} but compute the next conjugate vector from the given gradient g_k by adding a correction term. The basic algorithm for the CG method is given in Fig. 7.20.

7.4.2 Parallel CG Method

The parallel implementation of the CG method is based on the algorithm given in Fig. 7.20. Each iteration step of this algorithm implementing the CG method consists of the following basic vector and matrix operations.

7.4.2.1 Basic Operations of the CG Algorithm

The basic operations of the CG algorithm are

(1) a matrix–vector multiplication Ad_k,
(2) two scalar products $g_k^T g_k$ and $d_k^T w_k$,
(3) a so-called $axpy$-operation $x_k + \alpha_k d_k$
(The name $axpy$ comes from a x plus y describing the computation.),
(4) an $axpy$-operation $g_k + \alpha_k w_k$,
(5) a scalar product $g_{k+1}^T g_{k+1}$, and
(6) an $axpy$-operation $-g_{k+1} + \beta_k d_k$.

The result of $g_k^T g_k$ is needed in two consecutive steps and so the computation of one scalar product can be avoided by storing $g_k^T g_k$ in the scalar value γ_k. Since there are mainly one matrix–vector product and scalar products, a parallel implementation can be based on parallel versions of these operations.

Like the CG method many algorithms from linear algebra are built up from basic operations like matrix–vector operations or $axpy$-operations and efficient implementations of these basic operations lead to efficient implementations of the entire algorithms. The **BLAS** (*Basic Linear Algebra Subroutines*) library offers efficient implementations for a large set of basic operations. This includes many $axpy$-operations which denote that a vector x is multiplied by a scalar value a and then added to another vector y. The prefixes s in saxpy or d daxpy denote axpy-operations for *simple precision* and *double precision*, respectively. Introductory descriptions of the BLAS library are given in [43] or [60]. A standard way to parallelize algorithms for linear algebra is to provide efficient parallel implementations of the BLAS operations and to build up a parallel algorithm from these basic parallel

operations. This technique is ideally suited for the CG method since it consists of such basic operations.

Here, we consider a parallel implementation based on the parallel implementations for matrix–vector multiplication or scalar product for distributed memory machines as presented in Sect. 3. These parallel implementations are based on a data distribution of the matrix and the vectors involved. For an efficient implementation of the CG method it is important that the data distributions of different basic operations fit together in order to avoid expensive data re-distributions between the operations. Figure 7.21 shows a data dependence graph in which the nodes correspond to the computation steps (1)–(6) of the CG algorithm in Fig. 7.20 and the arrows depict a data dependency between two of these computation steps. The arrows are annotated with data structures computed in one step (outgoing arrow) and needed for another step with incoming arrow. The data dependence graph for one iteration step k is a directed acyclic graph (DAG). There are also data dependences to the previous iteration step $k-1$ and the next iteration step $k+1$, which are shown as dashed arrows.

There are the following dependences in the CG method: The computation (2) needs the result w_k from computation (1) but also the vector d_k and the scalar value γ_k from the previous iteration step $k-1$; γ_k is used to store the intermediate result $\gamma_k = g_k^T g_k$. Computation (3) needs α_k from computation step (2) and the vectors x_k, d_k from the previous iteration step $k-1$. Computation (4) also needs α_k from

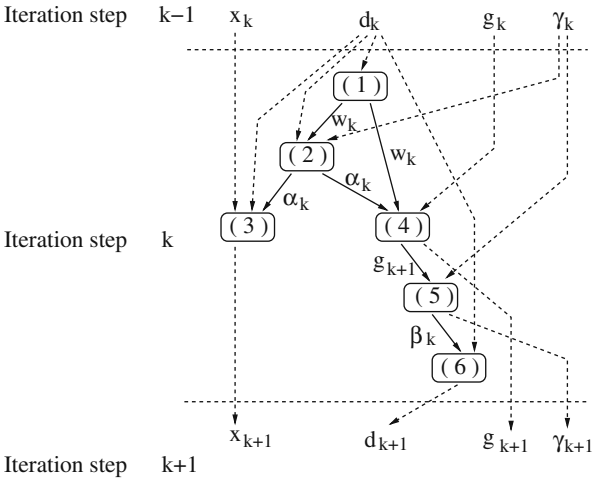


Fig. 7.21 Data dependences between the computation steps (1)–(6) of the CG method in Fig. 7.20. *Nodes* represent the computation steps of one iteration step k. *Incoming arrows* are annotated by the data required and *outgoing arrows* are annotated by the data produced. Two nodes have an *arrow* between them if one of the nodes produces data which are required by the node with the *incoming arrow*. The data dependences to the previous iteration step $k-1$ or the next iteration step $k+1$ are given as *dashed arrows*. The data are named in the same way as in Fig. 7.20; additionally the scalar γ_k is used for the intermediate result $\gamma_k = g_k^T g_k$ computed in step (5) and required for the computations of α_k and β_k in computation steps (2) and (5) of the next iteration step

computation step (2) and vector w_k from computation (1). Computation (5) needs vector g_{k+1} from computation (4) and scalar value γ_k from the previous iteration step $k-1$; computation (6) needs the scalar value from β_k from computation (5) and vector d_k from iteration step $k-1$. This shows that there are many data dependences between the different basic operations. But it can also be observed that computation (3) is independent of the computations (4)–(6). Thus, the computation sequence (1),(2),(3),(4),(5),(6) as well as the sequence (1),(2),(4),(5),(6),(3) can be used. The independence of computation (3) from computations (4)–(6) is also another source of parallelism, which is a coarse-grained parallelism of two linear algebra operations performed in parallel, in contrast to the fine-grained parallelism exploited for a single basic operation. In the following, we concentrate on the fine-grained parallelism of basic linear algebra operations.

When the basic operations are implemented on a distributed memory machine, the data distribution of matrices and vectors and the data dependences between operations might require data re-distribution for a correct implementation. Thus, the data dependence graph in Fig. 7.21 can also be used to study the communication requirements for re-distribution in a message-passing program. Also the data dependences between two iteration steps may lead to communication for data re-distribution.

To demonstrate the communication requirements, we consider an implementation of the CG method in which the matrix A has a row-blockwise distribution and the vectors d_k, ω_k, g_k, x_k, and r_k have a blockwise distribution. In one iteration step of a parallel implementation, the following computation and communication operations are performed.

7.4.2.2 Parallel CG Implementation with Blockwise Distribution

The parallel CG implementation has to consider data distributions in the following way:

(0) Before starting the computation of iteration step k, the vector d_k computed in the previous step has to be re-distributed from a blockwise distribution of step $k-1$ to a replicated distribution required for step k. This can be done with a multi-broadcast operation.

(1) The matrix–vector multiplication $w_k = Ad_k$ is implemented with a row-blockwise distribution of A as described in Sect. 3.6. Since d_k is now replicated, no further communication is needed. The result vector w_k is distributed in a blockwise way.

(2) The scalar product $d_k^T w_k$ is computed in parallel with the same blockwise distribution of both vectors. (The scalar product $\gamma_k = g_k^T g_k$ is computed in the previous iteration step.) Each processor computes a local scalar product for its local vectors. The final scalar product is then computed by the root processor of a single-accumulation operation with addition as reduction operation. This processor owns the final result α_k and sends it to all other processors by a single-broadcast operation.

(3) The scalar value α_k is known by each processor and thus the $axpy$-operation $x_{k+1} = x_k + \alpha_k d_k$ can be done in parallel without further communication. Each

processor performs the arithmetic operations locally and the vector x_{k+1} results in a blockwise distribution.

(4) The $axpy$-operation $g_{k+1} = g_k + \alpha_k w_k$ is computed analogously to computation step (3) and the result vector g_{k+1} is distributed in a blockwise way.

(5) The scalar product $\gamma_{k+1} = g_{k+1}^T g_{k+1}$ is computed analogously to computation step (2). The resulting scalar value β_k is computed by the root processor of a single-accumulation operation and then broadcasted to all other processors.

(6) The $axpy$-operation $d_{k+1} = -g_{k+1} + \beta_k d_k$ is computed analogously to computation step (3). The result vector d_{k+1} has a blockwise distribution.

7.4.2.3 Parallel Execution Time

The parallel execution time of one iteration step of the CG method is the sum of the parallel execution times of the basic operations involved. We derive the parallel execution time for p processors; n is the system size. It is assumed that n is a multiple of p. The parallel execution time of one $axpy$-operation is given by

$$T_{axpy} = 2 \cdot \frac{n}{p} \cdot t_{op} \ ,$$

since each processor computes n/p components and the computation of each component needs one multiplication and one addition. As in earlier sections, the time for one arithmetic operation is denoted by t_{op}. The parallel execution time of a scalar product is

$$T_{\text{scal\_prod}} = 2 \cdot \left(\frac{n}{p} - 1 \right) \cdot t_{op} + T_{\text{acc}}(+)(p, 1) + T_{sb}(p, 1) \ ,$$

where $T_{\text{acc}}(op)(p, m)$ denotes the communication time of a single-accumulation operation with reduction operation op on p processors and message size m. The computation of the local scalar products with n/p components requires n/p multiplications and $n/p - 1$ additions. The distribution of the result of the parallel scalar product, which is a scalar value, i.e., has size 1, needs the time of a single-broadcast operation $T_{sb}(p, 1)$. The matrix–vector multiplication needs time

$$T_{\text{math\_vec\_mult}} = 2 \cdot \frac{n^2}{p} \cdot t_{op} \ ,$$

since each processor computes n/p scalar products. The total computation time of the CG method is

$$T_{\text{CG}} = T_{\text{mb}} \left(p, \frac{n}{p} \right) + T_{\text{math\_vec\_mult}} + 2 \cdot T_{\text{scal\_prod}} + 3 \cdot T_{axpy},$$

where $T_{\text{mb}}(p, m)$ is the time of a multi-broadcast operation with p processors and message size m. This operation is needed for the re-distribution of the direction vector d_k from iteration step k .

7.5 Cholesky Factorization for Sparse Matrices

Linear equation systems arising in practice are often large but have sparse coefficient matrices, i.e., they have many zero entries. For sparse matrices with regular structure, like banded matrices, only the diagonals with non-zero elements are stored and the solution methods introduced in the previous sections can be used. For an unstructured pattern of non-zero elements in sparse matrices, however, a more general storage scheme is needed and other parallel solution methods are applied. In this section, we consider the Cholesky factorization as an example of such a solution method. The general sequential factorization algorithm and its variants for sparse matrices are introduced in Sect. 7.5.1. A specific storage scheme for sparse unstructured matrices is given in Sect. 7.5.2. In Sect. 7.5.3, we discuss parallel implementations of sparse Cholesky factorization for shared memory machines.

7.5.1 Sequential Algorithm

The Cholesky factorization is a direct solution method for a linear equation system $Ax = b$. The method can be used if the coefficient matrix $A = (a_{ij}) \in \mathbb{R}^{n\times n}$ is symmetric and positive definite, i.e., if $a_{ij} = a_{ji}$ and $x^T Ax > 0$ for all $x \in \mathbb{R}^n$ with $x \neq 0$. For a symmetric and positive definite $n \times n$ matrix $A \in \mathbb{R}^{n\times n}$ there exists a unique triangular factorization

$$A = LL^T, \tag{7.59}$$

where $L = (l_{ij})_{i,j=1,\ldots,n}$ is a lower triangular matrix, i.e., $l_{ij} = 0$ for $i < j$ and $i, j \in \{1, \ldots, n\}$, with positive diagonal elements, i.e., $l_{ii} > 0$ for $i = 1, \ldots, n$; L^T denotes the transposed matrix of L, i.e., $L^T = (l^T_{ij})_{i,j=1,\ldots,n}$ with $l^T_{ij} = l_{ji}$ [166]. Using the factorization in Eq. (7.59), the solution x of a system of equations $Ax = b$ with $b \in \mathbb{R}^n$ is determined in two steps by solving the triangular systems $Ly = b$ and $L^T x = y$ one after another. Because of $Ly = LL^T x = Ax = b$, the vector $x \in \mathbb{R}^n$ is the solution of the given linear equation system.

The implementation of the Cholesky factorization can be derived from a column-wise formulation of $A = LL^T$. Comparing the elements of A and LL^T, we obtain

$$a_{ij} = \sum_{k=1}^{n} l_{ik} l^T_{kj} = \sum_{k=1}^{n} l_{ik} l_{jk} = \sum_{k=1}^{j} l_{ik} l_{jk} = \sum_{k=1}^{j} l_{jk} l_{ik}$$

since $l_{jk} = 0$ for $k > j$ and by exchanging elements in the last summation. Denoting the columns of A as $\tilde{\mathbf{a}}_1, \ldots, \tilde{\mathbf{a}}_n$ and the columns of L as $\tilde{\mathbf{l}}_1, \ldots, \tilde{\mathbf{l}}_n$ results in an equality for column $\tilde{\mathbf{a}}_j = (a_{1j}, \ldots, a_{nj})$ and columns $\tilde{\mathbf{l}}_k = (l_{1k}, \ldots, l_{nk})$ for $k \leq j$:

$$\tilde{\mathbf{a}}_j = \sum_{k=1}^{j} l_{jk} \tilde{\mathbf{l}}_k$$

leading to

$$l_{jj} \tilde{\mathbf{l}}_j = \tilde{\mathbf{a}}_j - \sum_{k=1}^{j-1} l_{jk} \tilde{\mathbf{l}}_k \tag{7.60}$$

for $j = 1, \ldots, n$. If the columns $\tilde{\mathbf{l}}_k$, $k = 1, \ldots, j-1$, are already known, the right-hand side of Formula (7.60) is computable and the column $\tilde{\mathbf{l}}_j$ can also be computed. Thus, the columns of L are computed one after another. The computation of column $\tilde{\mathbf{l}}_j$ has two cases:

For the diagonal element the computation is

$$l_{jj} l_{jj} = a_{jj} - \sum_{k=1}^{j-1} l_{jk} l_{jk} \quad \text{or} \quad l_{jj} = \sqrt{a_{jj} - \sum_{k=1}^{j-1} l_{jk}^2}\,.$$

For the elements l_{ij}, $i > j$, the computation is

$$l_{ij} = \frac{1}{l_{jj}} \left(a_{ij} - \sum_{k=1}^{j-1} l_{jk} l_{ik} \right);$$

The elements in the upper triangular of matrix L are $l_{ij} = 0$ for $i < j$.

The Cholesky factorization yields the factorization $A = LL^T$ for a given matrix A [65] by computing $L = (l_{ij})_{i=0,\ldots,n-1,\, j=0,\ldots,i}$ from $A = (a_{ij})_{i,\, j=0,\ldots,n-1}$ column by column from left to right according to the following algorithm, in which the numbering starts with 0:

(I)

```
for (j=0; j<n; j++) {
    l_jj = sqrt( a_jj - sum_{k=0}^{j-1} l_jk^2 );
    for (i=j+1; i<n; i++)
        l_ij = 1/l_jj ( a_ij - sum_{k=0}^{j-1} l_jk l_ik );
}
```

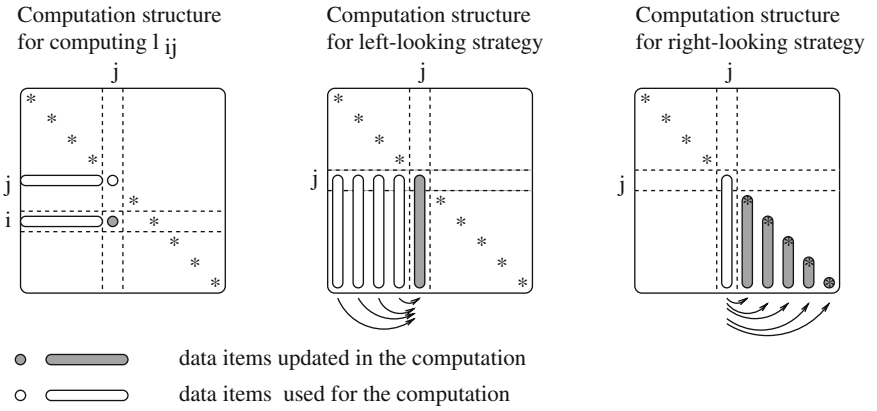


Fig. 7.22 Computational structures and data dependences for the computation of L according to the basic algorithm (*left*), the left-looking algorithm (*middle*), and the right-looking algorithm (*right*)

For each column j, first the new diagonal element l_{jj} is computed using the elements in row j; then, the new elements of column j are computed using row j of A and all columns i of L with $i < j$, see Fig. 7.22 (left).

For dense matrices A, the Cholesky factorization requires $O(n^2)$ storage space and $O(n^3/6)$ arithmetic operations [166]. For sparse matrices, drastic reductions in storage and execution time can be achieved by exploiting the sparsity of A, i.e., by storing and computing only the non-zero entries of A.

The Cholesky factorization usually causes fill-ins for sparse matrices A which means that the matrix L has non-zeros in positions which are zero in A. The number of fill-in elements can be reduced by reordering the rows and columns of A resulting in a matrix PAP^T with a corresponding permutation matrix P. For Cholesky factorization, P can be chosen without regard to numerical stability, because no pivoting is required [65]. Since PAP^T is also symmetric and positive definite for any permutation matrix P, the factorization of A can be done with the following steps:

1. **Reordering:** Find a permutation matrix $P \in \mathbb{R}^{n \times n}$ that minimizes the storage requirement and computing time by reducing fill-ins. The reordered linear equation system is $(PAP^T)(Px) = Pb$.
2. **Storage allocation:** Determine the structure of the matrix L and set up the sparse storage scheme. This is done before the actual computation of L and is called (*symbolic factorization*), see [65].
3. **Numerical factorization:** Perform the factorization $PAP^T = LL^T$.
4. **Triangular solution:** Solve $Ly = Pb$ and $L^T z = y$. Then, the solution of the original system is $x = P^T z$.

The problem of finding an ordering that minimizes the amount of fill-in is NP-complete [177]. But there exist suitable heuristics for reordering. The most

popular sequential fill-in reduction heuristic is the minimum degree algorithm [65]. Symbolic factorization by a graph-theoretic approach is described in detail in [65]. In the following, we concentrate on the numerical factorization, which is considered to require by far the most computation time, and assume that the coefficient matrix is already in reordered form.

7.5.1.1 Left-Looking Algorithms

According to [124], we denote the sparsity structure of column j and row i of L (excluding diagonal entries) by

$$Struct(L_{*j}) = \{k > j | l_{kj} \neq 0\}$$
$$Struct(L_{i*}) = \{k < i | l_{ik} \neq 0\}$$

$Struct(L_{*j})$ contains the row indices of all non-zeros of column j and $Struct(L_{i*})$ contains the column indices of all non-zeros of row i. Using these sparsity structures a slight modification of computation scheme (I) results. The modification uses the following procedures for manipulating columns [124, 152]:

(II)

```
cmod(j,k) =
    for each i ∈ Struct(L*k) with i ≥ j :
        a_ij = a_ij − l_jk l_ik ;
cdiv(j) =
    l_jj = √a_jj ;
    for each i ∈ Struct(L*j) :
        l_ij = a_ij / l_jj ;
```

Procedure $cmod(j, k)$ modifies column j by subtracting a multiple with factor l_{jk} of column k from column j for columns k already computed. Only the non-zero elements of column k are considered in the computation. The entries a_{ij} of the original matrix a are now used to store the intermediate results of the computation of L. Procedure $cdiv(j)$ computes the square root of the diagonal element and divides all entries of column j by this square root of its diagonal entry l_{jj}. Using these two procedures, column j can be computed by applying $cmod(j, k)$ for each $k \in Struct(L_{j*})$ and then completing the entries by applying $cdiv(j)$. Applying $cmod(j, k)$ to columns $k \notin Struct(L_{j*})$ has no effect because $l_{jk} = 0$. The columns of L are computed from left to right and the computation of a column $\tilde{l}_j$ needs all columns $\tilde{l}_k$ to the left of column $\tilde{l}_j$. This results in the following *left-looking* algorithm:

(III)

```
left_cholesky =
    for j = 0, ..., n − 1 {
        for each k ∈ Struct(L_{j*}):
            cmod(j, k);
        cdiv(j);
    }
```

The code in scheme (III) computes the columns one after another from left to right. The entries of column j are modified after all columns *to the left* of j have completely been computed, i.e., the same target column j is used for a number of consecutive $cmod(j, k)$ operations; this is illustrated in Fig. 7.22 (middle).

7.5.1.2 Right-Looking Algorithm

An alternative way is to use the entries of column j after the complete computation of column j to modify all columns k *to the right* of j that depend on column j, i.e., to modify all columns $k \in Struct(L_{*j})$ by subtracting l_{kj} times the column j from column k. Because $l_{kj} = 0$ for $k \notin Struct(L_{*j})$, only the columns $k \in Struct(L_{*j})$ are manipulated by column j. Still the columns are computed from left to right. The difference to the left-looking algorithm is that the calls to `cmod()` for a column j are done earlier. The final computation of a column j then consists only of a call to `cdiv`(j) after all columns to the left are computed. This results in the following *right-looking* algorithm:

(IV)

```
right_cholesky =
    for j = 0, ..., n − 1 {
        cdiv(j);
        for each k ∈ Struct(L_{*j}):
            cmod(k, j);
    }
```

The code fragment shows that in the right-looking algorithm, successive `cmod()` operations manipulate different target columns with the same column j. An illustration is given in Fig. 7.22 (right).

In both the left-looking and right-looking algorithms, each non-zero l_{ij} leads to an execution of a $cmod()$ operation. In the left-looking algorithm, the $cmod(j, k)$ operation is used to compute column j. In the right-looking algorithm, the $cmod(k, j)$ operation is used to manipulate column $k \in Struct(L_{*j})$ after the computation of column j. Thus, left-looking and right-looking algorithms use the same number of `cmod()` operations. They also use the same number of `cdiv()` operations, since there is exactly one `cdiv()` operation for each column.

7.5.1.3 Supernodes

The supernodal algorithm is a computation scheme for sparse Cholesky factorization that exploits similar patterns of non-zero elements in adjacent columns, see [124, 152]. A supernode is a set

$$I(p) = \{p,\, p+1, \ldots,\, p+q-1\}$$

of contiguous columns in L for which for all i with $p \le i \le p+q-1$

$$Struct(L_{*i}) = Struct(L_{*(p+q-1)}) \cup \{i+1, \ldots,\, p+q-1\}\ .$$

Thus, a supernode has a dense triangular block above (and including) row $p+q-1$, i.e., all entries are non-zero elements, and an identical sparsity structure for each column below row $p+q-1$, i.e., each column has its non-zero elements in the same rows as the other columns in the supernode. Figure 7.23 shows an example. Because of this identical sparsity structure of the columns, a supernode has the property that each member column modifies the same set of target columns outside its supernode [152]. Thus, the factorization can be expressed in terms of supernodes modifying columns, rather than columns modifying columns.

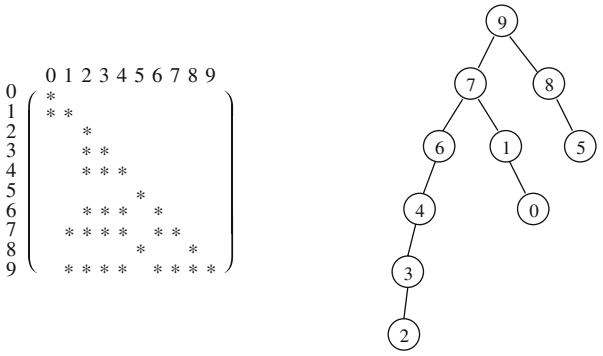


Fig. 7.23 Matrix L with supernodes $I(0) = \{0\}$, $I(1) = \{1\}$, $I(2) = \{2, 3, 4\}$, $I(5) = \{5\}$, $I(6) = \{6, 7\}$, $I(8) = \{8, 9\}$. The elimination tree is shown at the *right*

Using the definitions $first(J) = p$ and $last(J) = p+q-1$ for a supernode $J = I(p) = \{p, p+1, \ldots, p+q-1\}$, the following additional procedure $smod()$ is defined:

(V)

```
smod(j, J) =
      r = min{j − 1, last(J)};
      for k = first(J), ..., r
            cmod(j, k);
```

which modifies column j with all columns from supernode J. There are two cases for modifying a column with a supernode: When column j belongs to supernode J, then column j is modified only by those columns of J that are to the left in node J. When column j does not belong to supernode J, then column j is modified by all columns of J. Using the procedure $smod()$, the Cholesky factorization can be performed by the following computation scheme, also called *right-looking supernodal* algorithm:

(VI)

```
supernode_cholesky =
     for each supernode J do from left to right {
        cdiv(first(J));
        for j = first(J) + 1, ..., last(J) {
             smod(j, J);
             cdiv(j);
        }
        for k ∈ Struct(L*(last(J)))
             smod(k, J);
     }
```

This computation scheme still computes the columns of L from left to right. The difference to the algorithms presented before is that the computations associated with a supernode are combined. On the supernode level, a right-looking scheme is used: For the computation of the first column of a supernode J only one `cdiv()` operation is necessary when the modification with all columns to the left is already done. The columns of J are computed in a left-looking way: After the computation of all supernodes to the left of supernode J and because the columns of J are already modified with these supernodes due the supernodal right-looking scheme, column j is computed by first modifying it with all columns of J to the left of j and then performing a `cdiv()` operation. After the computation of all columns of J, all columns k to the right of J that depend on columns of J are modified with each column in J, i.e., by the procedure `smod`(k, J).

An alternative way would be a right-looking computation of the columns of J. An advantage of the supernodal algorithm lies in an increased locality of memory accesses because each column of a supernode J is used for the modification of several columns to the right of J and because all columns of J are used for the modification of the same columns to the right of J.

7.5.2 Storage Scheme for Sparse Matrices

Since most entries in a sparse matrix are zero, specific storage schemes are used to avoid the storage of zero elements. These compressed storage schemes store the non-zero entries and additional information about the row and column indices to

identify its original position in the full matrix. Thus, a compressed storage scheme for sparse matrices needs the space for the non-zero elements as well as space for additional information.

A sparse lower triangular matrix L is stored in a compressed storage scheme of size $O(n+nz)$ where n is the number of rows (or columns) in L and nz is the number of non-zeros. We present the storage scheme of the SPLASH implementation which, according to [116], stores a sparse matrix in a compressed manner similar to [64]. This storage scheme exploits the sparsity structure as well as the supernode structure to store the data. We first describe a simpler version using only the sparsity structure without supernodes. Exploiting the supernode structure is then based on this storage scheme.

The storage scheme uses two arrays `Nonzero` and `Row` of length nz and three arrays `StartColumn`, `StartRow`, and `Supernode` of length n. The array `Nonzero` contains the values of all non-zeros of a triangular matrix $L = (l_{kj})_{k \geq j}$ in column-major order, i.e., the non-zeros are ordered columnwise from left to right in a linear array. Information about the corresponding column indices of non-zero elements is implicitly contained in array `StartColumn`: Position j of array `StartColumn` stores the index of array `Nonzero` in which the first non-zero element of column j is stored, i.e., `Nonzero[StartColumn[`j`]]` contains l_{jj}. Because the non-zero elements are stored columnwise, `StartColumn[`$j+1$`]` -1 contains the last non-zero element of column j. Thus, the non-zeros of the jth column of L are assigned to the contiguous part of array `Nonzero` with indices from `StartColumn[`j`]` to `StartColumn[`$j+1$`]` -1. The size of the contiguous part of non-zeros of column j in array `Nonzero` is $N_j :=$ `StartColumn[`$j+1$`]`$-$`StartColumn[`j`]`. The array `Row` contains the row indices of the corresponding elements in `Nonzero`. In the simpler version without supernodes, `Row[`r`]` contains the row index of the non-zero stored in `Nonzero[`r`]`, $r = 0, \ldots, nz-1$. Corresponding to the blockwise storage scheme in `Nonzero`, the indices of the non-zeros of one column are stored in a contiguous block in `Row`.

When the similar sparsity structure of rows in the same supernode is additionally exploited, row indices of non-zeros are stored in a combination of the arrays `Row` and `StartRow` in the following way: `StartRow[`j`]` stores the index of `Row` in which the row index of the first non-zero of column j is stored, i.e., `Row[StartRow[`j`]]` = j because l_{jj} is the first non-zero. For each column the row indices are still stored in a contiguous block of `Row`. In contrast to the simpler scheme the blocks for different rows in the same supernode are not disjoint but overlap according to the similar sparsity structure of those columns.

The additional array `StartRow` can be used for a more compact storage scheme for the supernodal algorithm. When j is the first column of a supernode $I(j) = \{j, j+1, \ldots, j+k-1\}$, then column $j+l$ for $1 \leq l < k$ has the same non-zero pattern as row j for rows greater than or equal to $j+l$, i.e., `Row[StartRow[`j`]+`l`]` contains the row index of the first element of column $j+l$. Since this is the diagonal element, `Row[StartRow[`j`]+`l`]` $= j+l$ holds. The next entries are the row indices of the other non-zero elements of column $j+l$. Thus, the row indices of column $j+l$ are stored in `Row[StartRow[`j`]+`l`]`, $\ldots$, `Row[StartRow[`j`]+`

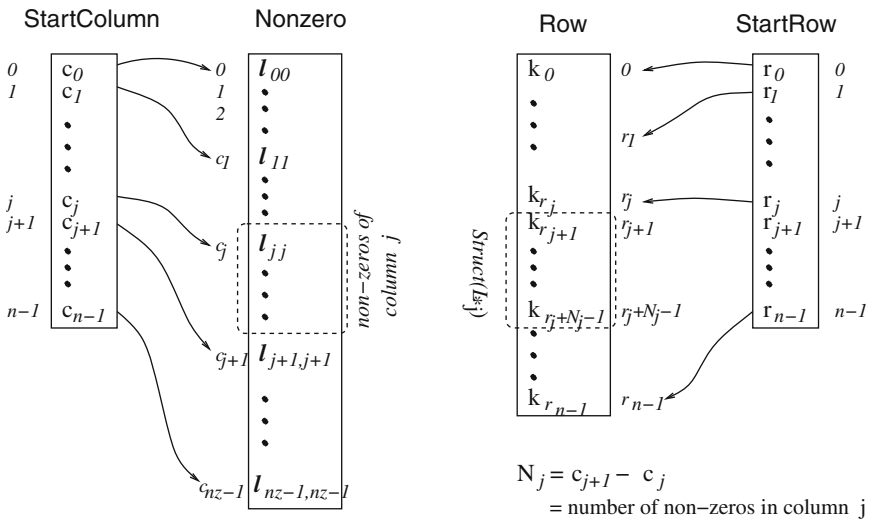


Fig. 7.24 Compressed storage scheme for a sparse lower triangular matrix L. The array `Nonzero` contains the non-zero elements of matrix L and the array `StartColumn` contains the positions of the first elements of columns in `Nonzero`. The array `Row` contains the row indices of elements in `Nonzero`; the first element of a row is given in `StartRow`. For a supernodal algorithm, `Row` can additionally use an overlapping storage (not shown here)

`StartColumn`$[j+1]-$`StartColumn`$[j-1]]$. This leads to `StartRow`$[j+l] =$ `StartRow`$[j]+l$ and thus only the row indices of the first column of a supernode have to be stored to get the full information. A fast access to the sets $Struct(L_{*j})$ is given by

$$Struct(L_{*j}) = \{\ \texttt{Row[StartRow}[j]+i] \mid 0 \le i \le \texttt{StartColumn}[j+1] - \texttt{StartColumn}[j-1]\}.$$

The storage scheme is illustrated in Fig. 7.24. The array `Supernode` is used for the management of supernodes: If a column j is the first column of a supernode J, then the number of columns of J is stored in `Supernode[j]`.

7.5.3 Implementation for Shared Variables

For a parallel implementation of sparse Cholesky factorization, we consider a shared memory machine. There are several sources of parallelism for sparse Cholesky factorization, including fine-grained parallelism within the single operations `cmod`(j,k) or `cdiv`(j) as well as column-oriented parallelism in the left-looking, right-looking, and supernodal algorithms.

The sparsity structure of L may lead to an additional source of parallelism which is not available for dense factorization. Data dependences may be avoided when different columns (and the columns having effect on them) have a disjoint sparsity structure. This kind of parallelism can be described by *elimination trees* that

express the specific situation of data dependences between columns using the relation $parent(j)$ [124, 118]. For each column j, $0 \leq j < n$, we define

$$parent(j) = min\{i \mid i \in Struct(L_{*j})\} \quad \text{if} \quad Struct(L_{*j}) \neq \emptyset,$$

i.e., $parent(j)$ is the row index of the first off-diagonal non-zero of column j. If $Struct(L_{*j}) = \emptyset$, then $parent(j) = j$. The element $parent(j)$ is the first column $i > j$ which depends on j. A column l, $j < l < i$, between them does not depend on j, since $j \notin Struct(L_{l*})$ and no $\texttt{cmod}(l, j)$ is executed. Moreover we define for $0 \leq i < n$

$$children(i) = \{j < i \mid parent(j) = i\},$$

i.e., $children(i)$ contains all columns j that have their first off-diagonal non-zero in row i.

The directed graph $G = (V, E)$ has a set of nodes $V = \{0, \ldots, n-1\}$ with one node for each column and a set of edges E, where $(i, j) \in E$ if $i = parent(j)$ and $i \neq j$. It can be shown that G is a tree if matrix A is *irreducible*. (A matrix A is called reducible if A can be permuted such that it is block-diagonal. For a reducible matrix, the blocks can be factorized independently.) In the following, we assume an irreducible matrix. Figure 7.25 shows a matrix and its corresponding elimination tree.

In the following, we denote the subtree with root j by $G[j]$. For sparse Cholesky factorization, an important property of the elimination tree G is that the tree specifies the order in which the columns must be evaluated: The definition of $parent$ implies that column i must be evaluated before column j, if $j = parent(i)$. Thus, all the children of column j must be completely evaluated before the computation of j. Moreover, column j does not depend on any column that is not in the subtree $G[j]$. Hence, columns i and j can be computed in parallel, if $G[i]$ and $G[j]$ are disjoint subtrees. Especially, all leaves of the elimination tree can be computed in parallel and the computation does not need to start with column 0. Thus, the sparsity structure determines the parallelism to be exploited. For a given matrix, elimination trees of smaller height usually represent a larger degree of parallelism than trees of larger height [77].

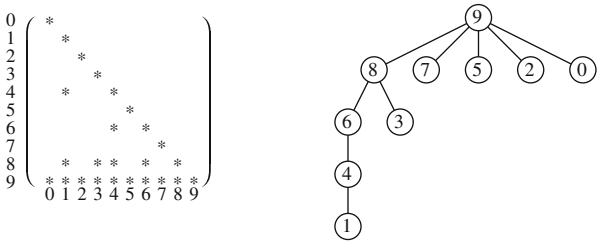


Fig. 7.25 Sparse matrix and the corresponding elimination tree

7.5.3.1 Parallel Left-Looking Algorithms

The parallel implementation of the left-looking algorithm (III) is based on n column tasks $Tcol(0), \ldots, Tcol(n-1)$ where task $Tcol(j)$, $0 \leq j < n$, comprises the execution of $\texttt{cmod}(j,k)$ for all $k \in Struct(L_{j*})$ and the execution of $\texttt{cdiv}(j)$; this is the loop body of the `for` loop in algorithm (III). These tasks are not independent of each other but have dependences due to the non-zero elements. The parallel implementation uses a task pool for managing the execution of the tasks. The task pool has a central task pool for storing column tasks, which can be accessed by every processor. Each processor is responsible for performing a subset of the column tasks. The assignment of tasks to processors for execution is dynamic, i.e., when a processor is idle, it takes a task from the central task pool.

The dynamic implementation has the advantage that the workload is distributed evenly although the tasks might have different execution times due to the sparsity structure. The concurrent accesses of the processors to the central task pool have to be conflict-free so that the unique assignment of a task to a processor for execution is guaranteed. This can be implemented by a locking mechanism so that only one processor accesses the task pool at a specific time.

There are several parallel implementation variants for the left-looking algorithm differing in the way the column tasks are inserted into the task pool. We consider three implementation variants:

- Variant L1 inserts column task $Tcol(j)$ into the task pool not before all column tasks $Tcol(k)$ with $k \in Struct(L_{j*})$ have been finished. The task pool can be initialized to the leaves of the elimination tree. The degree of parallelism is limited by the number of independent nodes of the tree, since tasks dependent on each other are executed in sequential order. Hence, a processor that has accessed task $Tcol(j)$ can execute the task without waiting for other tasks to be finished.
- Variant L2 allows to start the execution of $Tcol(j)$ without requiring that it can be executed to completion immediately. The task pool is initialized to all column tasks available. The column tasks are accessed by the processors dynamically from left to right, i.e., an idle processor accesses the next column that has not yet been assigned to a processor.

  The computation of column task $Tcol(j)$ is started before all tasks $Tcol(k)$ with $k \in Struct(L_{j*})$ have been finished. In this case, not all operations $\texttt{cmod}(j,k)$ of $Tcol(j)$ can be executed immediately but the task can perform only those $\texttt{cmod}(j,k)$ operations with $k \in Struct(L_{j*})$ for which the corresponding tasks have already been executed. Thus, the task might have to wait during its execution for other tasks to be finished.

  To control the execution of a single column task $Tcol(j)$, each column j is assigned a data structure S_j containing all columns $k \in Struct(L_{j*})$ for which $\texttt{cmod}(j,k)$ can already be executed. When a processor finishes the execution of the column task $Tcol(k)$ (by executing $\texttt{cdiv}(k)$), it pushes k onto the data structures S_j for each $j \in Struct(L_{*k})$. Because different processors might try to access the same stack at the same time, a locking mechanism has to be used to avoid access conflicts. The processor executing $Tcol(j)$ pops column indices

```
parallel_left_cholesky =
        c = 0;
        while ((j = get_unique_index()) < n) {
            for (i = 0; i < |Struct(L_j*)|; i++) {
                while (S_j empty) wait();
                k = pop(S_j);
                cmod(j, k);
            }
            cdiv(j);
            for (i ∈ Struct(L_*j)) : push(j, S_i);
        }
```

Fig. 7.26 Parallel left-looking algorithm according to variant L2. The implicit task pool is implemented in the `while` loop and the function `get_unique_index()`. The stacks $S_1, \ldots, S_n$ implement the bookkeeping about the dependent columns already finished

k from S_j and executes the corresponding `cmod`(j, k) operation. If S_j is empty, the processor waits for another processor to insert new column indices. When $|Struct(L_{j*})|$ column indices have been retrieved from S_j, the task $Tcol(j)$ can execute the final `cdiv`(j) operation.

Figure 7.26 shows the corresponding implementation. The central task pool is realized implicitly as a parallel loop; the operation `get_unique_index()` ensures a conflict-free assignment of tasks so that the processors accessing the pool at the same time get different unique loop indices representing column tasks. The loop body of the `while` loop implements one task $Tcol(j)$. The data structures $S_1, \ldots, S_n$ are stacks; pop(S_j) retrieves an element and push(j, S_i) inserts an element onto the stack.

- Variant L3 is a variation of L2 that takes the structure of the elimination tree into consideration. The columns are not assigned strictly from left to right to the processors, but according to their height in the elimination tree, i.e., the children of a column j in the elimination tree are assigned to processors before their parent j. This variant tries to complete the column tasks in the order in which the columns are needed for the completion of the other columns, thus exploiting the additional parallelism that is provided by the sparsity structure of the matrix.

7.5.3.2 Parallel Right-Looking Algorithm

The parallel implementation of the right-looking algorithm (IV) is also based on a task pool and on column tasks. These column tasks are defined differently than the tasks of the parallel left-looking algorithm: A column task $Tcol(j)$, $0 \leq j < n$, comprises the execution of `cdiv`(j) and `cmod`(k, j) for all $k \in Struct(L_{*j})$, i.e., a column task comprises the final computation for column j and the modifications of all columns $k > j$ right of column j that depend on j. The task pool is initialized to all column tasks corresponding to the leaves of the elimination tree. A task $Tcol(j)$ that is not a leaf is inserted into the task pool as soon as the operations `cmod`(j, k) for all $k \in Struct(L_{j*})$ are executed and a final `cdiv`(j) operation is possible.

Figure 7.27 sketches a parallel implementation of the right-looking algorithm. The task assignment is implemented by maintaining a counter c_j for each column j. The counter is initialized to 0 and is incremented after the execution of each `cmod`$(j, *)$ operation by the corresponding processor using the conflict-free

```
parallel_right_cholesky =
        c = 0;
        initialize_task_pool(TP);
        while ((j = get_unique_index())< n) {
            while (!filled_pool(TP,j)) wait();
            get_column(j);
            cdiv(j);
            for (k ∈ Struct(L*j)) {
                lock(k); cmod(k,j); unlock(k);
                if (add_counter(ck, 1) + 1 == |Struct(Lk*)|) add_column(k);
            }
        }
```

Fig. 7.27 Parallel right-looking algorithm. The column tasks are managed by a task pool `TP`. Column tasks are inserted into the task pool by `add_column()` and retrieved from the task pool by `get_column()`. The function `initialize_task_pool()` initializes the task pool `TP` with the leaves of the elimination tree. The condition of the outer `while` loop assigns column indices `j` to processors. The processor retrieves the corresponding column task as soon as the call `filled_pool(TP,j)` returns that the column task exists in the task pool

procedure `add_counter()`. For the execution of a `cmod`(k, j) operation of a task $Tcol(j)$, column k must be locked to prevent other tasks from modifying the same column at the same time. A task $Tcol(j)$ is inserted into the task pool, when the counter c_j has reached the value $|Struct(L_{j*})|$.

The differences between this right-looking implementation and the left-looking variant L2 lie in the execution order of the $cmod()$ operations and in the executing processor. In the L2 variant, the operation $cmod(j,k)$ is initiated by the processor computing column k by pushing it on stack S_j, but the operation is executed by the processor computing column j. This execution need not be performed immediately after the initiation of the operation. In the right-looking variant, the operation $cmod(j,k)$ is not only initiated, but also executed by the processor that computes column k.

7.5.3.3 Parallel Supernodal Algorithm

The parallel implementation of the supernodal algorithm uses a partition into *fundamental supernodes*. A supernode $I(p) = \{p, p+1, \ldots, p+q-1\}$ is a fundamental supernode, if for each i with $0 \leq i \leq q-2$, we have $children(p+i+1) = \{p+i\}$, i.e., node $p+i$ is the only child of $p+i+1$ in the elimination tree [124]. In Fig. 7.23, supernode $I(2) = \{2,3,4\}$ is a fundamental supernode whereas supernodes $I(6) = \{6,7\}$ and $I(8) = \{8,9\}$ are not fundamental. In a partition into fundamental supernodes, all columns of a supernode can be computed as soon as the first column can be computed and a waiting for the computation of columns outside the supernode is not needed. In the following, we assume that all supernodes are fundamental, which can be achieved by splitting supernodes into smaller ones. A supernode consisting of a single column is fundamental.

The parallel implementation of the supernodal algorithm (VI) is based on supernode tasks $Tsup(J)$ where task $Tsup(J)$ for $0 \leq J < N$ comprises the

execution of smod(j, J) and cdiv(j) for each $j \in J$ from left to right and the execution of smod(k, J) for all $k \in Struct(L_{*(last(J))})$; N is the number of supernodes. Tasks $Tsup(J)$ that are ready for execution are held in a central task pool that is accessed by the idle processors. The pool is initialized to supernodes that are ready for completion; these are those supernodes whose first column is a leaf in the elimination tree. If the execution of a cmod(k, J) operation by a task $Tsup(J)$ finishes the modification of another supernode $K > J$, the corresponding task $Tsup(K)$ is inserted into the task pool.

The assignment of supernode tasks is again implemented by maintaining a counter c_j for each column j of a supernode. Each counter is initialized to 0 and is incremented for each modification that is executed for column j. Ignoring the modifications with columns inside a supernode, a supernode task $Tsup(J)$ is ready for execution if the counters of the columns $j \in J$ reach the value $|Struct(L_{j*})|$. The implementation of the counters as well as the manipulation of the columns has to be protected, e.g., by a lock mechanism. For the manipulation of a column $k \notin J$ by a smod(k, J) operation, column k is locked to avoid concurrent manipulation by different processors. Figure 7.28 shows the corresponding implementation.

```
parallel_supernode_cholesky =
        c = 0;
        initialize_task_pool(TP);
        while ((J = get_unique_index()) < N) {
            while (!filled_pool(TP,J)) wait();
            get_supernode(J);
            cdiv(first(J));
            for (j = first(J) + 1; j ≤ last(J); j++) {
                smod(j,J);
                cdiv(j);
            }
            for (k ∈ Struct(L_{*(last(J))})) {
                lock(k); smod(k,J); unlock(k);
                add_counter(c_k, 1);
                if ((k == first(K)) && (c_k + 1 == |Struct(L_{k*})|))
                    add_supernode(K);
            }
        }
```

Fig. 7.28 Parallel supernodal algorithm

7.6 Exercises for Chap. 7

Exercise 7.1 For an $n \times m$ matrix A and vectors a and b of length n write a parallel MPI program which computes a rank-1 update $A = A - a \cdot b^T$ which can be computed sequentially by

```
for (i=0; i<n; i++)
   for (j=0; j<n; j++)
      A[i][j] = A[i][j]-a[i] · b[j];
```

For the parallel MPI implementation assume that A is distributed among the p processors in a column-cyclic way. The vectors a and b are available at the process with rank 0 only and must be distributed appropriately before the computation. After the update operation, the matrix A should again be distributed in a column-cyclic way.

Exercise 7.2 Implement the rank-1 update in OpenMP. Use a parallel `for` loop to express the parallel execution.

Exercise 7.3 Extend the program piece in Fig. 7.2 for performing the Gaussian elimination with a row-cyclic data distribution to a full MPI program. To do so, all helper functions used and described in the text must be implemented. Measure the resulting execution times for different matrix sizes and different numbers of processors.

Exercise 7.4 Similar to the previous exercise, transform the program piece in Fig. 7.6 with a total cyclic data distribution to a full MPI program. Compare the resulting execution times for different matrix sizes and different numbers of processors. For which scenarios does a significant difference occur? Try to explain the observed behavior.

Exercise 7.5 Develop a parallel implementation of Gaussian elimination for shared address spaces using OpenMP. The MPI implementation from Fig. 7.2 can be used as an orientation. Explain how the available parallelism is expressed in your OpenMP implementation. Also explain where synchronization is needed when accessing shared data. Measure the resulting execution times for different matrix sizes and different numbers of processors.

Exercise 7.6 Develop a parallel implementation of Gaussian elimination using Java threads. Define a new class `Gaussian` which is structured similar to the Java program in Fig. 6.23 for a matrix multiplication. Explain which synchronization is needed in the program. Measure the resulting execution times for different matrix sizes and different numbers of processors.

Exercise 7.7 Develop a parallel MPI program for Gaussian elimination using a column-cyclic data distribution. An implementation with a row-cyclic distribution has been given in Fig. 7.2. Explain which communication is needed for a column-cyclic distribution and include this communication in your program. Compute the resulting speedup values for different matrix sizes and different numbers of processors.

Exercise 7.8 For $n = 8$ consider the following tridiagonal equation system:

$$\begin{pmatrix} 1 & 1 & & & & \\ 1 & 2 & 1 & & & \\ & 1 & 2 & \ddots & & \\ & & \ddots & \ddots & \ddots & 1 \\ & & & & 1 & 2 \end{pmatrix} \cdot x = \begin{pmatrix} 1 \\ 2 \\ 3 \\ \vdots \\ 8 \end{pmatrix}.$$

Use the recursive doubling technique from Sect. 7.2.2, p. 385, to solve this equation system.

Exercise 7.9 Develop a sequential implementation of the cyclic reduction algorithm for solving tridiagonal equation systems, see Sect. 7.2.2, p. 385. Measure the resulting sequential execution times for different matrix sizes starting with size $n = 100$ up to size $n = 10^7$.

Exercise 7.10 Transform the sequential implementation of the cyclic reduction algorithm from the last exercise into a parallel implementation for a shared address space using OpenMP. Use an appropriate parallel `for` loop to express the parallel execution. Measure the resulting parallel execution times for different numbers of processors for the same matrix sizes as in the previous exercise. Compute the resulting speedup values and show the speedup values in a diagram.

Exercise 7.11 Develop a parallel MPI implementation of the cyclic reduction algorithm for a distributed address space based on the description in Sect. 7.2.2, p. 385. Measure the resulting parallel execution times for different numbers of processors and compute the resulting speedup values.

Exercise 7.12 Specify the data dependence graph for the cyclic reduction algorithm for $n = 12$ equations according to Fig. 7.11. For $p = 3$ processors, illustrate the three phases according to Fig. 7.12 and show which dependences lead to communication.

Exercise 7.13 Implement a parallel Jacobi iteration with a pointer-based storage scheme of the matrix A such that global indices are used in the implementation.

Exercise 7.14 Consider the parallel implementation of the Jacobi iteration in Fig. 7.13 and provide a corresponding shared memory program using OpenMP operations.

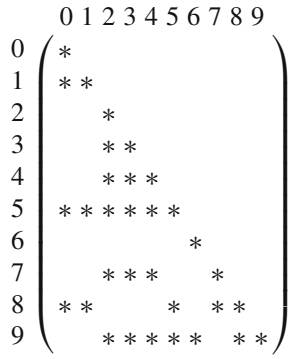
Exercise 7.15 Implement a parallel SOR method for a dense linear equation system by modifying the parallel program in Fig. 7.14.

Exercise 7.16 Provide a shared memory implementation of the Gauss–Seidel method for the discretized Poisson equation.

Exercise 7.17 Develop a shared memory implementation for Cholesky factorization $A = LL^T$ for a dense matrix A using the basic algorithm.

Exercise 7.18 Develop a message-passing implementation for dense Cholesky factorization $A = LL^T$.

Exercise 7.19 Consider a matrix with the following non-zero entries:



(a) Specify all supernodes of this matrix.
(b) Consider a supernode J with at least three entries. Specify the sequence of *cmod* and *cdiv* operations that are executed for this supernode in the right-looking supernode Cholesky factorization algorithm.
(c) Determine the elimination tree resulting for this matrix.
(d) Explain the role of the elimination tree for a parallel execution.

Exercise 7.20 Derive the parallel execution time of a message-passing program of the CG method for a distributed memory machine with a linear array as interconnection network.

Exercise 7.21 Consider a parallel implementation of the CG method in which the computation step (3) is executed in parallel to the computation step (4). Given a row-blockwise distribution of matrix A and a blockwise distribution of the vector, derive the data distributions for this implementation variant and give the corresponding parallel execution time.

Exercise 7.22 Implement the CG algorithm given in Fig. 7.20 with the blockwise distribution as message-passing program using MPI.

References

1. F. Abolhassan, J. Keller, and W.J. Paul. On the Cost–Effectiveness of PRAMs. In *Proceedings of the 3rd IEEE Symposium on Parallel and Distributed Processing*, pages 2–9, 1991.
2. A. Adl-Tabatabai, C. Kozyrakis, and B. Saha. Unlocking concurrency. *ACM Queue*, 4(10): 24–33, Dec 2006.
3. S.V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29: 66–76, 1995.
4. A. Aggarwal, A.K. Chandra, and M. Snir. On Communication Latency in PRAM Computations. In *Proceedings of 1989 ACM Symposium on Parallel Algorithms and Architectures* (*SPAA'89*), pages 11–21, 1989.
5. A. Aggarwal, A.K. Chandra, and M. Snir. Communication complexity of PRAMs. *Theoretical Computer Science*, 71: 3–28, 1990.
6. A. Aho, M. Lam, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques & Tools*. Pearson-Addison Wesley, Boston, 2007.
7. A.V. Aho, M.S. Lam, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, and Tools*. 2nd Edition, Addison-Wesley Longman Publishing Co., Inc., Boston, 2006.
8. S.G. Akl. *Parallel Computation – Models and Methods*. Prentice Hall, Upper Saddle River, 1997.
9. K. Al-Tawil and C.A. Moritz. LogGP Performance Evaluation of MPI. *International Symposium on High-Performance Distributed Computing,* page 366, 1998.
10. A. Alexandrov, M. Ionescu, K.E. Schauser, and C. Scheiman. LogGP: Incorporating Long Messages into the LogP Model – One Step Closer Towards a Realistic Model for Parallel Computation. In *Proceedings of the 7th ACM Symposium on Parallel Algorithms and Architectures* (*SPAA'95*), pages 95–105, Santa Barbara, July 1995.
11. E. Allen, D. Chase, J. Hallett, V. Luchangco, J.-W. Maessen, S. Ryu, G.L. Steele, Jr., and S. Tobin-Hochstadt. *The Fortress Language Specification, version 1.0 beta*, March 2007.
12. R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann, San Francisco, 2002.
13. S. Allmann, T. Rauber, and G. Rünger. Cyclic Reduction on Distributed Shared Memory Machines. In *Proceedings of the 9th Euromicro Workshop on Parallel and Distributed Processing*, pages 290–297, Mantova, Italien, 2001. IEEE Computer Society Press.
14. G.S. Almasi and A. Gottlieb. *Highly Parallel Computing*. Benjamin Cummings, New York, 1994.
15. G. Amdahl. Validity of the Single Processor Approach to Achieving Large-Scale Computer Capabilities. In *AFIPS Conference Proceedings*, volume 30, pages 483–485, 1967.
16. K. Asanovic, R. Bodik, B.C. Catanzaro, J.J. Gebis, P. Husbands, K. Keutzer, D.A. Patterson, W.L. Plishker, J. Shalf, S.W. Williams, and K.A. Yelick. The Landscape of Parallel Computing Research: A View from Berkeley. Technical Report UCB/EECS-2006–183, EECS Department, University of California, Berkeley, December 2006.

DOI 10.1007/978-3-642-04818-0,

17. M. Azimi, N. Cherukuri, D.N. Jayasimha, A. Kumar, P. Kundu, S. Park, I. Schoinas, and A. Vaidya. Integration challenges and tradeoffs for tera-scale architectures. *Intel Technology Journal*, 11(03), 2007.
18. C.J. Beckmann and C. Polychronopoulos. Microarchitecture Support for Dynamic Scheduling of Acyclic Task Graphs. Technical Report CSRD 1207, University of Illinois, 1992.
19. D.P. Bertsekas and J.N. Tsitsiklis. *Parallel and Distributed Computation*. Athena Scientific, Nashua, 1997.
20. R. Bird. *Introduction to Functional Programming Using Haskell*. Prentice Hall, Englewood Cliffs, 1998.
21. P. Bishop and N. Warren. *JavaSpaces in Practice*. Addison Wesley, Reading, 2002.
22. F. Bodin, P. Beckmann, D.B. Gannon, S. Narayana, and S. Yang. Distributed C++: Basic Ideas for an Object Parallel Language. In *Proceedings of the Supercomputing'91 Conference*, pages 273–282, 1991.
23. D. Braess. *Finite Elements*. 3rd edition, Cambridge University Press, Cambridge, 2007.
24. P. Brucker. *Scheduling Algorithms*. 4th edition, Springer-Verlag, Berlin, 2004.
25. D.R. Butenhof. *Programming with POSIX Threads*. Addison-Wesley Longman Publishing Co., Inc., Boston, 1997.
26. N. Carriero and D. Gelernter. Linda in Context. *Communications of the ACM*, 32(4): 444–458, 1989.
27. N. Carriero and D. Gelernter. *How to Write Parallel Programs*. MIT Press, Cambridge, 1990.
28. P. Charles, C. Grothoff, V.A. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: An Object-Oriented Approach to Non-uniform Cluster Computing. In R. Johnson and R.P. Gabriel, editors, *Proceedings of the 20th Annual ACMSIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (*OOPSLA*), pages 519–538, ACM, October 2005.
29. A. Chin. Complexity Models for All-Purpose Parallel Computation. In *Lectures on Parallel Computation*, chapter 14. Cambridge University Press, Cambridge, 1993.
30. M.E. Conway. A Multiprocessor System Design. In *Proceedings of the AFIPS 1963 Fall Joint Computer Conference*, volume 24, pages 139–146, Spartan Books, New York, 963.
31. T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, 2001.
32. D.E. Culler, A.C. Arpaci-Dusseau, S.C. Goldstein, A. Krishnamurthy, S. Lumetta, T. van Eicken, and K.A. Yelick. Parallel Programming in Split-C. In *Proceedings of Supercomputing*, pages 262–273, 1993.
33. D.E. Culler, A.C. Dusseau, R.P. Martin, and K.E. Schauser. Fast Parallel Sorting Under LogP: From Theory to Practice. In *Portability and Performance for Parallel Processing*, pages 71–98. Wiley, Southampton, 1994.
34. D.E. Culler, R. Karp, A. Sahay, K.E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a Realistic Model of Parallel Computation. *Proceedings of the 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (*PPoPP'93*), pages 1–12, 1993.
35. D.E. Culler, J.P. Singh, and A. Gupta. *Parallel Computer Architecture: A Hardware Software Approach*. Morgan Kaufmann, San Francisco, 1999.
36. H.J. Curnov and B.A. Wichmann. A Synthetic Benchmark. *The Computer Journal*, 19(1): 43–49, 1976.
37. D. Callahan, B.L. Chamberlain, and H.P. Zima. The Cascade High Productivity Language. In *IPDPS*, pages 52–60. IEEE Computer Society, 2004.
38. W.J. Dally and C.L. Seitz. Deadlock-Free Message Routing in Multiprocessor Interconnection Networks. *IEEE Transactions on Computers*, 36(5): 547–553, 1987.
39. DEC. The Whetstone Performance. Technical Report, Digital Equipment Corporation, 1986.
40. E.W. Dijkstra. Cooperating Sequential Processes. In F. Genuys, editor, *Programming Languages*, pages 43–112. Academic Press Inc., 1968.

41. J. Dongarra. Performance of various Computers using Standard Linear Equations Software in Fortran Environment. Technical Report CS-89–85, Computer Science Department, University of Tennessee, Knoxville, 1990.
42. J. Dongarra and W. Gentzsch, editors. *Computer Benchmarks*. Elsevier, North Holland, 1993.
43. J.J. Dongarra, I.S. Duff, D.C. Sorenson, and H.A. van der Vorst. *Solving Linear Systems on Vector and Shared Memory Computers*. SIAM, Philadelphia, 1993.
44. J. Duato, S. Yalamanchili, and L. Ni. *Interconnection Networks – An Engineering Approach*. Morgan Kaufmann, San Francisco, 2003.
45. M. Dubois, C. Scheurich, and F. Briggs. Memory Access Buffering in Multiprocessors. In *Proceedings of the 13th International Symposium on Computer Architecture* (*ISCA'86*), pages 434–442, ACM, 1986.
46. J. Dümmler, T. Rauber, and G. Rünger. Mixed Programming Models using Parallel Tasks. In J. Dongarra, C.-H. Hsu, K.-C. Li, L.T. Yang, and H. Zima, editors, *Handbook of Research on Scalable Computing Technologies*. Information Science Reference, July 2009.
47. T. El-Ghazawi, W. Carlson, T. Sterling, and K. Yelick. *UPC: Distributed Shared Memory Programming*. Wiley, New York, 2005.
48. J.R. Ellis. *Bulldog: A Compiler for VLIW Architectures*. MIT Press, Cambridge, MA, USA, 1986.
49. T. Ellis, I. Phillips, and T. Lahey. *Fortran90 Programming*. Addison-Wesley, Wokingham, 1994.
50. J.T. Feo. An analysis of the computational and parallel complexity of the Livermore loops. *Parallel Computing*, 7: 163–185, 1988.
51. D. Flanagan. *Java in a Nutshell*. O'Reilly, Sebastopol, 2005.
52. M.J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, 21(9): 948–960, 1972.
53. S. Fortune and J. Wyllie. Parallelism in Random Access Machines. In *Proceedings of the 10th ACM Symposium on Theory of Computing*, pages 114–118, 1978.
54. High Performance Fortran Forum. High performance Fortran language specification. *Scientific Programming*, 2(1): 1–165, 1993.
55. Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard, Version 1.3*. www.mpi-forum.org, 2008.
56. Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard, Version 2.1*. www.mpi-forum.org, 2008.
57. I. Foster. *Designing and Building Parallel Programs*. Addison-Wesley, Reading, 1995.
58. I. Foster. Compositional parallel programming languages. *ACM Transactions on Programming Languages and Systems*, 18(4): 454–476, 1996.
59. I. Foster. Globus Toolkit Version 4: Software for Service-Oriented Systems. In *Proceedings of the IFIP International Conference on Network and Parallel Computing*, pages 2–13, Springer LNCS 3779, 2006.
60. T.L. Freeman and C. Phillips. *Parallel Numerical Algorithms*. Prentice Hall, Upper Saddle River, 1992.
61. A. Frommer. *Lösung linearer Gleichungssysteme auf Parallelrechnern*. Vieweg, Braunschweig, 1990.
62. M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman, New York, 1979.
63. A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM Parallel Virtual Machine: A User's Guide and Tutorial for Networked Parallel Computing*. MIT Press, Cambridge, 1996. Web page: www.netlib.org/pvm3/book/pvm book.html.
64. A. George, J. Liu, and E. Ng. User's Guide for SPARSPAK: Waterloo Sparse Linear Equations Package. Technical Report CS-78–30, Department of Computer Science, University of Waterloo, 1980.
65. A. George and J.W.-H. Liu. *Computer Solution of Large Sparse Positive Definite Systems*. PrenticeHall, Englewood Cliffs, 1981.

66. P.B. Gibbons. A More Practical PRAM Model. In *Proceedings of the 1989 ACM Symposium on Parallel Algorithms and Architectures* (*SPAA'89*), pages 158–168, 1989.
67. M.B. Girkar and C. Polychronopoulos. Automatic extraction of functional parallelism from ordinary programs. *IEEE Transactions on Parallel and Distributed Systems*, 3(2): 166–178, 1992.
68. C.J. Glass and L.M. Li. The Turn Model for Adaptive Routing. In *Proceedings of the 19th International Symposium on Computer Architecture* (*ISCA'92*), pages 278–287, ACM, 1992.
69. S. Goedecker and A. Hoisie. *Performance Optimization of Numerically Intensive Codes*. SIAM, Philadelphia, 2001.
70. B. Goetz. *Java Concurrency in Practice*. Addison Wesley, Reading, 2006.
71. G. Golub and Ch. Van Loan. *Matrix Computations*. 3rd edition, The Johns Hopkins University Press, Baltimore, 1996.
72. G. Golub and J. Ortega. *Scientific Computing*. Academic Press, Boston, 1993.
73. A. Gottlieb, R. Grishman, C. Kruskal, K. McAuliffe, L. Rudolph, and M. Snir. The NYU ultracomputer – designing an MIMD shared memory parallel computer. *IEEE Transactions on Computers*, 32(2): 175–189, February 1983.
74. M.W. Goudreau, J.M. Hill, K. Lang, W.F. McColl, S.D. Rao, D.C. Stefanescu, T. Suel, and T. Tsantilas. A proposal for a BSP Worldwide standard. Technical Report, BSP Worldwide, www.bsp-worldwide.org, 1996.
75. A. Grame, A. Gupta, G. Karypis, and V. Kumar. *Introduction to Parallel Programming*. Addison Wesley, Reading, 2003.
76. T. Grün, T. Rauber, and J. Röhrig. Support for efficient programming on the SB-PRAM. *International Journal of Parallel Programming*, 26(3): 209–240, 1998.
77. A. Gupta, G. Karypis, and V. Kumar. Highly scalable parallel algorithms for sparse matrix factorization. *IEEE Transactions on Parallel and Distributed Systems*, 8(5): 502–520, 1997.
78. J.L. Gustafson. Reevaluating Amdahl's law. *Communications of the ACM*, 31(5): 532–533, 1988.
79. W. Hackbusch. *Iterative Solution of Large Sparse Systems of Equations*. Springer, New York, 1994.
80. K. Hammond and G. Michaelson, editors. *Research Directions in Parallel Functional Programming*. Springer-Verlag, Springer, 1999.
81. J. Handy. *The Cache Memory Book*. 2nd edition, Academic Press, San Diego, 1998.
82. P.J. Hatcher and M.J. Quinn. *Data-Parallel Programming*. MIT Press, Cambridge, 1991.
83. J. Held, J. Bautista, and S. Koehl. From a Few Cores to Many – A Tera-Scale Computing Research Overview. Intel White Paper, Intel, 2006.
84. J.L. Hennessy and D.A. Patterson. *Computer Architecture – A Quantitative Approach*. 4th edition, Morgan Kaufmann, Boston, 2007.
85. M. Herlihy and J.E.B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture* (*ISCA'93*), pages 289–300, 1993.
86. M.R. Hestenes and E. Stiefel. Methods of conjugate gradients for solving linear systems. *Journal of Research of the National Bureau of Standards*, 49: 409–436, 1952.
87. T. Heywood and S. Ranka. A practical hierarchical model of parallel computation. *Journal of Parallel and Distributed Computing*, 16: 212–249, 1992.
88. J.M.D. Hill, B. McColl, D.C. Stefanescu, M.W. Goudreau, K. Lang, S.B. Rao, T. Suel, T. Tsantilas, and R. Bisseling. BSPlib The BSB Programming Library. Technical Report TR-29–97, Oxford University, May 1997.
89. M. Hill, W. McColl, and D. Skillicorn. Questions and answers about BSP. *Scientific Programming*, 6(3): 249–274, 1997.
90. C.A.R. Hoare. Monitors: An operating systems structuring concept. *Communications of the ACM*, 17(10): 549–557, 1974.
91. R. Hockney. A fast direct solution of Poisson's equation using Fourier analysis. *Journal of the ACM*, 12: 95–113, 1965.

92. R.W. Hockney. *The Science of Computer Benchmarking*. SIAM, Philadelphia, 1996.
93. R. Hoffmann and T. Rauber. Fine-Grained Task Scheduling using Adaptive Data Structures. In *Proceedings of the of Euro-Par*, volume 5168 of Lecture Notes in Computer Science, pages 253–262, Springer, 2008.
94. P. Hudak and J. Fasel. A gentle introduction to Haskell. *ACM SIGPLAN Notices*, 27(5): May 1992.
95. K. Hwang. *Advanced Computer Architecture: Parallelism, Scalability, Programmability*. McGraw-Hill, New York, 1993.
96. F. Ino, N. Fujimoto, and K. Hagihara. LogGPS: A Parallel Computational Model for Synchronization Analysis. In *PPoPP '01: Proceedings of the Eighth ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, pages 133–142, ACM, New York, 2001.
97. J.D. Jackson. *Classical Electrodynamics*. 3rd edition, Wiley, New York and Chichester, 1998.
98. J. Jájá. *An Introduction to Parallel Algorithms*. Addison-Wesley, New York, 1992.
99. M. Johnson. *Superscalar Microprocessor Design*. Prentice Hall, Englewood Cliffs, 1991.
100. S. Johnsson and C. Ho. Optimum broadcasting and personalized communication in hypercubes. *IEEE Transactions on Computers*, 38(9): 1249–1268, 1989.
101. J. Keller, C.W. Keßler, and J.L. Träff. *Practical PRAM Programming*. Wiley, New York, 2001.
102. J. Keller, T. Rauber, and B. Rederlechner. Conservative Circuit Simulation on Shared–Memory Multiprocessors. In *Proceedings of the 10th Workshop on Parallel and Distributed Simulation* (*PADS'96*), pages 126–134, ACM, 1996.
103. K. Kennedy, C. Koelbel, and H. Zima. The Rise and Fall of High Performance Fortran: An Historical Object Lesson. In *HOPL III: Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*, pages 7–1–7–22, ACM, New York, 2007.
104. T. Kielmann, H.E. Bal, and K. Verstoep. Fast Measurement of LogP Parameters for Message Passing Platforms. In *IPDPS '00: Proceedings of the 15 IPDPS 2000 Workshops on Parallel and Distributed Processing*, pages 1176–1183, Springer, London, 2000.
105. S. Kleiman, D. Shah, and B. Smaalders. *Programming with Threads*. Prentice Hall, Englewood Cliffs, 1996.
106. G. Koch. Discovering Multi-core: Extending the Benefits of Moore's Law. Intel White Paper, Technology@Intel Magazine, 2005.
107. P.M. Kogge. An Exploitation of the Technology Space for Multi-Core Memory/Logic Chips for Highly Scalable Parallel Systems. In *Proceedings of the Innovative Architecture for Future Generation High-Performance Processors and Systems*, IEEE, 2005.
108. M. Korch and T. Rauber. A comparison of task pools for dynamic load balancing of irregular algorithms. *Concurrency and Computation: Practice and Experience*, 16: 1–47, January 2004.
109. D. Kuck. Platform 2015 Software-Enabling Innovation in Parallelism for the Next Decade. Intel White Paper, Technology@Intel Magazine, 2005.
110. J. Kurose and K. Ross. *Computer Networking, 3. Auflage*. Addison Wesley, Wokingham, 2005.
111. L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 28(9): 690–691, September 1979.
112. J.R. Laurs and R. Rajwar. *Transactional Memory*. Morgan & Claypool Publishers, San Rafael, 2007.
113. D. Lea. *Concurrent Programming in Java: Design Principles and Patterns*. Addison Wesley, Reading, 1999.
114. E.A. Lee. The problem with threads. *IEEE Computer*, 39(5): 33–42, 2006.
115. F.T. Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*. Morgan Kaufmann, San Mateo, 1992.
116. D.E. Lenoski and W.Weber. *Scalable Shared-Memory Multiprocessing*. Morgan Kaufmann, San Francisco, 1995.

117. B. Lewis and D.J. Berg. *Multithreaded Programming with Pthreads*. Prentice Hall, New Jersey, 1998.
118. J.W.H. Liu. The role of elimination trees in sparse factorization. *The SIAM Journal on Matrix Analysis and Applications*, 11: 134–172, 1990.
119. D.T. Marr, F. Binus, D.L. Hill, G. Hinton, D.A. Konfaty, J.A. Miller, and M. Upton. Hyper-threading technology architecture and microarchitecture. *Intel Technology Journal*, 6(1): 4–15, 2002.
120. T. Mattson, B. Sandor, and B. Massingill. *Pattern for Parallel Programming*. Pearson – Addison Wesley, Reading, 2005.
121. F. McMahon. The Livermore Fortran Kernels: A Computer Test of the Numerical Performance Range. Technical Report UCRL-53745, Lawrence Livermore National Laboratory, Livermore, 1986.
122. M. Metcalf and J. Reid. *Fortran 90/95 Explained*. Oxford University Press, Oxford, 2002.
123. R. Miller and L. Boxer. *Algorithms Sequential and Parallel*. Prentice Hall, Upper Saddle River, 2000.
124. E.G. Ng and B.W. Peyton. A Supernodal Cholesky Factorization Algorithm for Shared-Memory Multiprocessors. Technical Report, Oak Ridge National Laboratory, 1991.
125. L.M. Ni and P.K. McKinley. A survey of wormhole routing techniques in direct networks. *IEEE Computer*, 26: 62–76, February 1993.
126. B. Nichols, D. Buttlar, and J. Proulx Farrell. *Pthreads Programming*. O'Reilly & Associates, Sebastopol, 1997.
127. J. Nieplocha, J. Ju, M.K. Krishnan, B. Palmer, and V. Tipparaju. The Global Arrays User's Manual. Technical Report PNNL-13130, Pacific Northwest National Laboratory, 2002.
128. Nvidia. NVIDIA GeForce 8800 GPU Architecture Overview. Technical Report TB-02787–001 v01, Nvidia, 2006.
129. S. Oaks and H. Wong. *Java Threads. 3. Auflage*. O'Reilly, Sebastopol, 2004.
130. *OpenMP C and C++ Application Program Interface, Version 1.0*. www.openmp.org, October 1998.
131. *OpenMP Application Program Interface, Version 2.5*. www.openmp.org, May 2005.
132. *OpenMP Application Program Interface, Version 3.0*. www.openmp.org, May 2008.
133. J.M. Ortega. *Introduction to Parallel and Vector Solutions of Linear Systems*. Plenum Publishing Corp., New York, 1988.
134. J.M. Ortega and R.G. Voigt. *Solution of Partial Differential Equations on Vector and Parallel Computers*. SIAM, Philadelphia, 1985.
135. P.S. Pacheco. *Parallel Programming with MPI*. Morgan Kaufmann, San Francisco, 1997.
136. C.H. Papadimitriou and M. Yannakakis. Towards an Architecture-Independent Analysis of Parallel Algorithms. In *Proceedings of the 20th ACM Symposium on Theory of Computing*, pages 510–513, 1988.
137. D.A. Patterson and J.L. Hennessy. *Computer Organization & Design – The Hardware/Software Interface*. 4th edition, Morgan Kaufmann, San Francisco, 2008.
138. S. Pelegatti. *Structured Development of Parallel Programs*. Taylor and Francis, London, 1998.
139. L. Peterson and B. Davie. *Computer Networks – A Systems Approach*, *3. Auflage*. Morgan Kaufmann, Los Altos, 2003.
140. G.F. Pfister. *In Search of Clusters*. 2nd edition, Prentice Hall, Upper Saddle River, 1998.
141. A. Podehl, T. Rauber, and G. Rünger. A shared-memory implementation of the hierarchical radiosity method. *Theoretical Computer Science*, 196(1–2): 215–240, 1998.
142. C.D. Polychronopoulos. *Parallel Programming and Compilers*. Kluwer Academic Publishers, Norwell, 1988.
143. S. Prasad. *Multithreading Programming Techniques*. McGraw-Hill, New York, 1997.
144. R. Rajwar and J. Goodman. Transactional execution: Towards reliable, high-performance multithreading. *IEEE Micro*, 23(6): 117–125, 2003.

145. S. Ramaswamy, S. Sapatnekar, and P. Banerjee. A framework for exploiting task and data parallelism on distributed-memory multicomputers. *IEEE Transactions on Parallel and Distributed Systems*, 8(11): 1098–1116, 1997.
146. T. Rauber and G. Rünger. A transformation approach to derive efficient parallel implementations. *IEEE Transactions on Software Engineering*, 26(4): 315–339, 2000.
147. T. Rauber and G. Rünger. Deriving array distributions by optimization techniques. *Journal of Supercomputing*, 15: 271–293, 2000.
148. T. Rauber and G. Rünger. Tlib – A library to support programming with hierarchical multiprocessor tasks. *Journal of Parallel and Distributed Computing*, 65(3): 347–360, 2005.
149. T. Rauber, G. Rünger, and C. Scholtes. Execution behavior analysis and performance prediction for a shared-memory implementation of an irregular particle simulation method. *Simulation: Practice and Theory*, 6: 665–687, 1998.
150. J.K. Reid. On the Method of Conjugate Gradients for the Solution of Large Sparse Systems of Linear Equations. In *Large Sparse Sets of Linear Equations*, pages 231–254. Academic Press, New York, 1971.
151. M. Rosing, R.B. Schnabel, and R.P. Waever. The DINO Parallel Programming Language. Technical Report CU-CS-501–90, Computer Science Dept., University of Colorado at Boulder, Boulder, 1990.
152. E. Rothberg and A. Gupta. An evaluation of left-looking, right-looking and multifrontal approaches to sparse Cholesky factorization on hierarchical-memory machines. *International Journal of High Speed Computing*, 5(4): 537–593, 1993.
153. G. Rünger. Parallel Programming Models for Irregular Algorithms. In *Parallel Algorithms and Cluster Computing*, pages 3–23. Springer Lecture Notes in Computational Science and Engineering, 2006.
154. Y. Saad. *Iterative Methods for Sparse Linear Systems*. International Thomson Publ., London, 1996.
155. Y. Saad. Krylov subspace methods on supercomputers. *SIAM Journal on Scientific and Statistical Computing*, 10: 1200–1332, 1998.
156. J. Savage. *Models of Computation: Exploring the Power of Computing*. Addison-Wesley Longman Publishing Co., Inc., Boston, 1997.
157. C. Scheurich and M. Dubois. Correct Memory Operation of Cache-Based Multiprocessors. In *Proceedings of the 14th International Symposium on Computer Architecture* (*ISCA'87*), pages 234–243, ACM, 1987.
158. D. Sima, T. Fountain, and P. Kacsuk. *Advanced Computer Architectures*. Addison-Wesley, Harlow, 1997.
159. J.P. Singh. *Parallel Hierarchical N-Body Methods and Their Implication for Multiprocessors*. PhD Thesis, Stanford University, 1993.
160. D. Skillicorn and D. Talia. Models and languages for parallel computation. *ACM Computing Surveys*, 30(2): 123–169, 1998.
161. B. Smith. Architecture and applications on the HEP multiprocessor computer systems. *SPIE* (*Real Time Signal Processing IV*), 298: 241–248, 1981.
162. M. Snir, S. Otto, S. Huss-Ledermann, D. Walker, and J. Dongarra. *MPI: The Complete Reference*. MIT Press, Cambridge, 1996. Web page: www.netlib.org/utk/papers/mpi book/mpi book.html.
163. M. Snir, S. Otto, S. Huss-Ledermann, D. Walker, and J. Dongarra. *MPI: The Complete Reference, Vol. 1: The MPI Core*. MIT Press, Cambridge, 1998. Web page: mitpress.mit.edu/catalog/item/default.asp?ttype=2&tid=4579.
164. W. Stallings. *Computer Organization and Architecture*. 7th edition, Prentice Hall, Upper Saddle River, 2009.
165. R.C. Steinke and G.J. Nutt. A unified theory of shared memory consistency. *Journal of the ACM*, 51(5): 800–849, 2004.
166. J. Stoer and R. Bulirsch. *Introduction to Numerical Analysis*. Springer, New York, 2002.

167. H.S. Stone. Parallel processing with the perfect shuffle. *IEEE Transactions on Computers*, 20(2): 153–161, 1971.
168. H.S. Stone. An efficient parallel algorithm for the solution of a tridiagonal linear system of equations. *Journal of the ACM*, 20: 27–38, 1973.
169. H. Sutter and J. Larus. Software and the concurrency revolution. *ACM Queue*, 3(7): 54–62, 2005.
170. S. Thompson. *Haskell – The Craft of Functional Programming*. Addison-Wesley, Reading, 1999.
171. L.G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8): 103–111, 1990.
172. L.G. Valiant. A Bridging Model for Multi-core Computing. In *Proceedings of the ESA*, volume 5193, pages 13–28, Springer LNCS, 2008.
173. E.F. van de Velde. *Concurrent Scientific Computing*. Springer, New York, 1994.
174. R.P. Weicker. Dhrystone: A synthetic system programming benchmark. *Communications of the ACM*, 29(10): 1013–1030, 1984.
175. M. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley, Redwood City, 1996.
176. Xelerated. Xelerator X11 Network Processor. Technical report, Xelerated, www.xelerated.com, accessed 2009.
177. M. Yannakakis. Computing the minimum fill-in is NP-complete. *SIAM Journal on Algebraic and Discrete Methods*, 2: 77–79, 1991.
178. S.N. Zheltov and S.V. Bratanov. Measuring HT-Enabled Multi-Core: Advantages of a Thread-Oriented Approach. *Technology & Intel Magazine*, December 2005.
179. A.Y.H. Zomaya, editor. *Parallel & Distributed Computing Handbook*. Computer Engineering Series. McGraw-Hill, New York, 1996.

Index