Roman Trobec, Boštjan Slivnik,
Patricio Bulić, Borut Robič

Introduction to Parallel Computing: From Algorithms to Programming on State-of-the-Art Platforms

– Monograph –

July 2, 2018

Springer

To all who make our lives worthwhile.

Preface

This monograph is an overview of practical parallel computing and starts with the basic principles and rules which will enable the reader to design efficient parallel programs for solving various computational problems on the state-of-the-art computing platforms.

The book too was written in parallel. The opening Chapter 1: "Why do we need parallel programming" has been shaped by all of us during instant communication immediately after the idea of writing such a book had cropped up. In fact, the first chapter was an important motivation for our joint work. We spared no effort in incorporating of our teaching experience into this book.

The book consists of three parts: Foundations, Programming, and Engineering, each with a specific focus:

- Part I, **Foundations**, provides the *motivation* for embarking on a study of parallel computation (Chapter 1) and an *introduction* to parallel computing (Chapter 2) that covers parallel computer systems, the role of communication, complexity of parallel problem-solving, and the associated principles and laws.
- Part II, **Programming**, first discusses *shared-memory* platforms and openMP (Chapter 3), then proceeds to *message passing* library (Chapter 4), and finally to *massively parallel* processors (Chapter 5). Each chapter describes the methodology and practical examples for immediate work on a personal computer.
- Part III, **Engineering**, *illustrates* parallel solving of computational problems on three selected problems from three fields: *Computing the number* π (Chapter 6) from mathematics, *Solving the heat equation* (Chapter 7) from physics, and *Seam carving* (Chapter 8) from computer science. The book concludes with some final remarks and perspectives (Chapter 9).

To *enable readers to immediately start gaining practice* in parallel computing, **Appendix A** provides hints for making a personal computer ready to execute parallel programs under Linux, macOS, and MS Windows.

Specific contributions of the authors are as follows:

- *Roman Trobec* started the idea of writing a practical textbook, useful for students and programmers on a basic and advanced levels. He has contributed Chapter 4: "MPI processes and messaging", Chapter 9: "Final remarks and perspectives", and to chapters of Part III.
- *Boštjan Slivnik* has contributed Chapter 3: "Programming multi-core and shared memory multiprocessors using OpenMP". He has also contributed to Chapter 1 and chapters of Part III.
- *Patricio Bulić* has contributed Chapter 5: "OpenCL for massively parallel graphic processors" and Chapter 8: "Engineering: Parallel implementation of Seam Carving." His contribution is also in Chapter 1 and chapters of Part III.
- *Borut Robič* has coordinated our work and cared about the consistency of the book text. He has contributed Chapter 2: "Overview of parallel systems" and to Chapter 1: "Why do we need parallel programming."

The spectrum of book topics ranges from implementations to efficient applications of parallel processors on different platforms. The book covers shared memory many-core processors, shared memory multi-core processors, and interconnected distributed computers. Chapters of Parts I and II are quite independent and can be read in any order, while chapters of Part III are related to previous chapters and are intended to be a final reading.

The target audience comprises undergraduate and graduate students; engineers, programmers and industrial experts acting in companies that develop software with an intention to add parallel capabilities for increased performance; research institutions that develop and test computationally intensive software with parallel software codes; universities and educational institutions that teach courses on parallel computing. The book may also be interesting and useful for the wider public in the part where basic principles and benefits of parallel approaches are presented.

For the readers who wish to be promptly updated with current achievements in the field of parallel computing, we will maintain this information on the book web page. There, also a pool of questions and homeworks will be available and maintained according to experiences and feedbacks from readers.

We are grateful to all our colleagues who have contributed to this book through discussions or by reading and commenting parts of the text, in particular to Matjaž Depolli for his assistance in testing the exemplar programs, and to Andrej Brodnik for his fruitful suggestions and comments.

For their support of our work, we are indebted to the Jožef Stefan Institute, Faculty of Computer and Information Science of the University of Ljubljana, and the Slovenian Research Agency.

Ljubljana,
June 2018

Roman Trobec, Boštjan Slivnik,
Patricio Bulić, Borut Robič

Contents

Part I
Foundations

In Part I, we first provide the motivation for delving into the realm of parallel computation and especially of parallel programming. There are several reasons for doing so: first, our computers are already parallel; secondly, parallelism can be of great practical value when it comes to solving computationally demanding problems from various areas; and finally, there is inertia in the design of contemporary computers which keeps parallelism a key ingredient of future computers.

The second chapter provides an introduction to parallel computing. It describes different parallel computer systems and formal models for describing such systems. Then, various patterns for interconnecting processors and memories are described and the role of communication is emphasized. All these issues have impact on the execution time required to solve computational problems. Thus, we introduce the necessary topics of the parallel computational complexity. Finally, we present some laws and principles that govern parallel computation.

Chapter 1
Why do we need parallel programming

Abstract The aim of this chapter is to give a motivation for the study of parallel computing and in particular parallel programming. Contemporary computers are parallel and there are various reasons for that. Parallelism comes in three different prevailing types which share common underlying principles. Most importantly, parallelism can help us solve demanding computational problems.

1.1 Why - every computer is a parallel computer

Nowadays all computers are essentially *parallel.* This means that within every operating computer there always exist various activities which, one way or another, run in parallel, *at the same time*. Parallel activities may arise and come to an end independently of each other—or, they may be *created purposely* to involve simultaneous performance of various operations whose interplay will eventually lead to the desired result. Informally, the parallelism is the existence of parallel activities within a computer and their use in achieving a common goal. The parallelism is found on all levels of a modern computer's architecture:

- First, parallelism is present deep in the **processor microarchitecture**. In the past, processors ran programs by repeating the so-called *instruction cycle*, a sequence of four steps: (*i*) reading and decoding an instruction; (*ii*) finding data needed to process the instruction; (*iii*) processing the instruction; and (*iv*) writing the result out. Since step (*ii*) introduced lengthy delays which were due to the arriving data, much of research focused on designs that reduced these delays and in this way increased the effective execution speed of programs. Over the years, however, the main goal has become the design of a processor capable of execution of *several instructions simultaneously*. The workings of such a processor enabled detection and exploitation of parallelism inherent in instruction execution. These processors allowed even higher execution speeds of programs, regardless of the processor and memory frequency.

- Second, any commercial computer, tablet and smartphone contains a processor with **multiple cores**, each of which is capable of running its own instruction stream. If the streams are designed so that the cores collaborate in running an application, the application is run in parallel and may be considerably sped up.
- Third, many servers contain a **several multicore processors**. Such a server is capable of running a service in parallel, and also several services in parallel.
- Finally, even consumer level computers contain **graphic processors** capable of running hundreds or even thousands of threads in parallel. Processors capable of coping with such a large parallelism are necessary to support graphic animation.

There are many reasons for making modern computers parallel:

- First, it is not possible to increase processor and memory **frequencies** indefinitely, at least not with the current silicon-based technology. Therefore, to increase computational power of computers new architectural and organizational *concepts* are needed.
- Second, **power consumption** rises with processor frequency while the energy efficiency decreases. However, if the computation is performed in parallel at lower processor speed, the undesirable implications of frequency increase can be avoided.
- Finally, parallelism has become a part of any computer and this is likely to remain unchanged due to simple **inertia**: parallelism *can* be done and it sells *well*.

1.2 How - there are three prevailing types of parallelism

During the last decades, many different parallel computing systems appeared on the market. First they have been sold as supercomputers dedicated to solving specific scientific problems. Perhaps the most known are the computers made by Cray and Connection Machine Corporation. But as mentioned above, the parallelism has spread all the way down into the consumer market and all kinds of handheld devices.

Various parallel solutions gradually evolved into modern parallel systems that exhibit at least one of the three prevailing types of parallelism:

- First, **shared memory systems**, i.e., systems with multiple processing units attached to a single memory.
- Second, **distributed systems**, i.e., systems consisting of many computer units, each with its own processing unit and its physical memory, that are connected with fast interconnection networks.
- Third, **graphic processor units** used as co-processors for solving general purpose numerically intensive problems.

Apart from the parallel computer systems that have become ubiquitous, extremely powerful **supercomputers** continue to dominate the parallel computing achievements. Supercomputers can be found on the **Top 500** list of the fastest computer systems ever built and even today they are the joy and pride of the world superpowers.

But the underlying principles of parallel computing are the same regardless of whether the top supercomputers or consumer devices are being programmed. The **programming principles and techniques** gradually evolved during all these years. Nevertheless, the design of parallel algorithms and parallel programming are still considered to be an order of magnitude harder than the design of sequential algorithms and sequential-program development.

Relating to the three types of parallelism introduced above, three different approaches to parallel programming exist: **threads** model for shared memory systems, **message passing** model for distributed systems, and **stream** based model for GPUs.

1.3 What - time consuming computations can be sped up

To see how parallelism can help you solve problems, it is best to look at examples. In this section, we will briefly discuss the so-called n-body problem.

The n-body problem

The *classical n-body problem* is the problem of predicting the individual motions of a group of objects that interact with each other by gravitation. Here is a more accurate statement of the problem:

The classical n-body problem

Given the position and momentum of each member of a group of bodies at an initial instant, compute their positions and velocities for all future instances.

While the classical n-body problem was motivated by the desire to understand the motions of the Sun, Moon, planets and the visible stars, it is nowadays used to comprehend the dynamics of globular cluster star systems. In this case, the usual Newton mechanics, which governs the moving of bodies, must be replaced by the Einsteins general relativity theory, which makes the problem even more difficult. We will, therefore, refrain from dealing with this version of the problem and focus on the classical version as introduced above and on the way it is solved on a parallel computer.

So how can we solve a given classical n-body problem? Let us first describe in what form we expect the solution of the problem. As mentioned above, the classical n-body problem assumes the classical, Newton's mechanics, which we all learned in school. Using this mechanics, a given instance of the n-body problem is described as a particular system of $6n$ differential equations that, for each of n bodies, define its location $(x(t), y(t), z(t))$ and momentum $(mv_x(t), mv_y(t), mv_z(t))$ at an instant t.

The solution of this system is the sought-for description of the evolution of the n-body system at hand. Thus, the question of solvability of a particular classical n-body problem boils down to the question of solvability of the associated system of differential equations that are finally transformed into a system of linear equations.

Today we know that

- if $n = 2$, the classical n-body problem always has analytical solution, simply because the associated system of equations has an analytic solution.
- if $n > 2$, analytic solutions exist *just for certain* initial configurations of n bodies.
- *In general, however, n-body problems cannot be solved analytically*.

It follows that, in general, the n-body problem must be solved *numerically*, by using appropriate numerical methods for solving systems of differential equations.

Can we always succeed in this? The numerical methods numerically integrate the differential equations of motion. To obtain the solution, such methods require time which grows proportionally to n^2. We say that the methods have time complexity of the order $O(n^2)$. At first sight, this seems to be rather promising; however, there is a *large hidden factor* in this $O(n^2)$. Because of this factor, only the instances of the n-body problem with small values of n can be solved using these numerical methods. To extend solvability to larger values of n, methods with smaller time complexity must be found. One such is the Barnes-Hut method with time complexity $O(n \log n)$. But, again, only the instances with limited (though larger) values of n can be solved. *For large values of n, numerical methods become prohibitively time consuming*.

Unfortunately, the values of n are in practice usually very large. Actually, they are *too large* for the above mentioned numerical methods to be of any practical value.

What can we do in this situation? Well, at this point **parallel computation** enters the stage. The numerical methods which we use for solving systems of differential equations associated with the n-body problem are usually programmed for *single*-processor computers. But if we have at our disposal a **parallel computer** with many processors, it is natural to consider using all of them so that they collaborate and *jointly* solve systems of differential equations. To achieve that, however, we must answer several **nontrivial questions**: (i) How can we partition a given numerical method into subtasks? (ii) Which subtasks should each processor perform? (iii) How should each processor collaborate with other processors? And then, of course, (iv) How will we code all of these answers in the form of a **parallel program**, a program capable of running on the parallel computer and exploiting its resources.

The above questions are not easy, to be sure, but there *have been* designed parallel algorithms for the above numerical methods, and written parallel programs that implement the algorithms for different parallel computers. For example, J. Dubinsky et al. designed a parallel Barnes-Hut algorithm and parallel program which divides the n-body system into independent rectangular volumes each of which is mapped to a processor of a parallel computer. The parallel program was able to simulate evolution of n-body systems consisting of $n = 640000$ to $n = 1100000$ bodies. It turned out that, for such systems, the optimal number of processing units was 64. At that number the processors were best load-balanced and communication between them was minimal.

1.4 And this book - why would you read it?

We believe that this book could provide the fist step in the process of attaining the ability to efficiently solve, on a *parallel computer*, not only the n-body problem but also many other computational problems of a myriad of scientific and applied problems whose high computational and/or data complexities make them virtually intractable even on the *fastest sequential computers*.

Chapter 2
Overview of parallel systems

Abstract In this chapter we overview the most important basic notions, concepts, and theoretical results concerning parallel computation.

2.1 History of parallel computing, systems and programming

Let Π be an arbitrary computational problem which is to be solved by a computer. Usually our first objective is to design an *algorithm* for solving Π. Clearly, the class of all algorithms is infinite, but we can partition it into two subclasses, the class of all sequential algorithms and the class of all parallel algorithms.[1] While a *sequential algorithm* performs one operation in each step, a **parallel algorithm** may perform multiple operations in a single step. In this book, we will be mainly interested in parallel algorithms. So, our objective is to design a parallel algorithm for Π.

Let P be an arbitrary parallel algorithm. We say that there is **parallelism** in P. The parallelism in P can be exploited by various *kinds* of **parallel computers**. For instance, multiple operations of P may be executed simultaneously by multiple **processing units** of a parallel computer C_1; or, perhaps, they may be executed by multiple pipelined functional units of a single-processor computer C_2. After all, P can always be *sequentially* executed on a single-processor computer C_3, simply by executing P's potentially parallel operations one by one in succession.

Let $C(p)$ be a parallel computer of the kind C which contains p processing units. Naturally, we expect the **performance** of P on $C(p)$ to depend both on C and p. We must, therefore, clearly distinguish between the **potential parallelism** in P on the one side, and the **actual capability** of $C(p)$ to execute, in parallel, multiple operations of P, on the other side. So the performance of the algorithm P on the parallel computer $C(p)$ depends on $C(p)$'s capability to exploit P's potential parallelism.

[1] There are also other divisions that partition the class of all algorithms according to other criteria, such as *exact* and *non-exact* algorithms; or *deterministic* and *non-deterministic* algorithms. However, in this book we will not divide algorithms systematically according to these criteria.

Before we continue, we must unambiguously define what we really mean by the term “performance” of a parallel algorithm P. Intuitively, the “performance" might mean the time required to execute P on $C(p)$; this is called the **parallel execution time** (or, **parallel runtime**) of P on $C(p)$, which we will denote by

$$T_{\text{par}}.$$

Alternatively, we might choose the “performance” to mean how many times is the parallel execution of P on $C(p)$ faster than the sequential execution of P; this is called the **speedup** of P on $C(p)$,

$$S \stackrel{\text{def}}{=} \frac{T_{\text{seq}}}{T_{\text{par}}}.$$

So parallel execution of P on $C(p)$ is S-times faster than sequential execution of P. Next, we might be interested in how much of the speedup S is, on average, due to each of the processing units. Put differently, the term “performance” might be understood as the average contribution of each of the p processing units of $C(p)$ to the speedup; this is called the **efficiency** of P on $C(p)$,

$$E \stackrel{\text{def}}{=} \frac{S}{p}.$$

Since $T_{\text{par}} \leqslant T_{\text{seq}} \leqslant p \cdot T_{\text{par}}$, it follows that speedup is bounded above by and efficiency is bounded above by

$$E \leqslant 1.$$

This means that, for *any C and p*, the parallel execution of P on $C(p)$ can be at most p times faster than the execution of P on a single processor. And the efficiency of the parallel execution of P on $C(p)$ can be at most 1. (This is when each processing unit is continually engaged in the execution of P, thus contributing $\frac{1}{p}$-th to its speedup.) Later, in Section 2.5, we will involve one more parameter to these definitions.

From the above definitions we see that both speedup and efficiency depend on T_{par}, the parallel execution time of P on $C(p)$. This raises new questions:

How do we determine T_{par}?
How does T_{par} depend on C (the kind of a parallel computer) ?
Which properties of C must we take into account in order to determine T_{par}?

These are important general questions about parallel computation which must be answered prior to embarking on a practical design and analysis of parallel algorithms. The way to answer these questions is to appropriately **model** parallel computation.

2.2 Modeling parallel computation

Parallel computers vary greatly in their organization. We will see in the next section that their processing units may or may not be directly connected one to another; some of the processing units may share a common memory while the others may only own local (private) memories; the operation of the processing units may be synchronized by a common clock, or they may run each at its own pace. Furthermore, usually there are architectural details and hardware specifics of the components, all of which show up during the actual design and use of a computer. And finally, there are technological differences, which manifest in different clock rates, memory access times etc. Hence, the following question arises:

Which properties of parallel computers **must be considered**
and which **may be ignored** *in the design and analysis of parallel algorithms?*

To answer the question, we apply ideas similar to those discovered in the case of sequential computation. There, various **models of computation** were discovered.[2] In short, the intention of each of these models was to *abstract the relevant* properties of the (sequential) computation *from the irrelevant* ones.

In our case, a model called the **Random Access Machine (RAM)** is particularly attractive. Why? The reason is that RAM distills the important properties of the general-purpose sequential computers, which are still extensively used today, and which have actually been taken as the *conceptual basis for modeling of parallel computing and parallel computers*. Figure 2.1 shows the structure of the RAM.

Here is a brief description of RAM:

- The RAM consists of a **processing unit** and a **memory**. The *memory* is a potentially infinite sequence of equally sized locations $m_0, m_1, \ldots$. The index i is called the *address* of m_i. Each location is directly accessible by the processing unit: given an arbitrary i, reading from m_i or writing to m_i is accomplished in constant time. *Registers* are a sequence $r_1 \ldots r_n$ of locations in the processing unit. Registers are directly accessible. Two of them have special roles. *Program counter pc* ($= r_1$) contains the address of the location in the memory which contains the instruction to be executed next. *Accumulator a* ($= r_2$) is involved in the execution of each instruction. Other registers are given roles as needed. The *program* is a finite sequence of instructions (similar to those in real computers).
- Before the RAM is started, the following is done: (a) a program is loaded into successive locations of the memory starting with, say, m_0; (b) input data are written into empty memory locations, say after the last program instruction.

[2] Some of these models of computation are the μ-recursive functions, recursive functions, λ-calculus, Turing machine, Post machine, Markov algorithms, and RAM.

- From now on, the RAM operates independently in a mechanical stepwise fashion as instructed by the program. Let $pc = k$ at the beginning of a step. (Initially, $k = 0$.) From the location m_k, the instruction `I` is read and started. At the same time, pc is incremented. So, when `I` is completed, the next instruction to be executed is in m_{k+1}, unless `I` was one of the instructions that change pc (e.g. jump instructions).

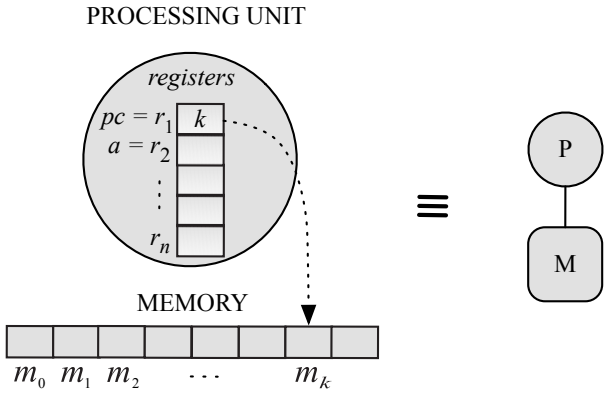


Fig. 2.1: The RAM model of computation has a memory M (containing program instructions and data) and a processing unit P (executing instructions on data).

So the above question boils down to the following question:

What is the **appropriate model** *of parallel computation?*

It turned out that finding an answer to this question is substantially more challenging than it was in the case of sequential computation. Why? Since there are many ways to organize parallel computers, there are also many ways to model them; and what is difficult is to select a *single* model that will be appropriate for *all* parallel computers.

As a result, in the last decades, researchers proposed *several* models of parallel computation. However, no common agreement has been reached about which is the right one. In the following, we describe those that are based on RAM.[3]

[3] In fact, currently the research is being pursued also in other, non-conventional directions, which do not build on RAM or any other conventional computational models (listed in previous footnote). Such are, for example, *dataflow computation* and *quantum computation*.

2.3 Multiprocessor models

A **multiprocessor model** is a model of parallel computation that builds on the RAM model of computation; that is, it generalizes the RAM. How does it do that?

It turns out that the generalization can be done in three essentially different ways resulting in three different multiprocessor models. Each of the three models has some number $p\,(\geqslant 2)$ of processing units, but the models differ in the organization of their memories and in the way the processing units access the memories.

The models are called the

- *Parallel Random Access Machine* (*PRAM*),
- *Local Memory Machine* (*LMM*), and
- *Modular Memory Machine* (*MMM*).

Let us describe them.

2.3.1 The Parallel Random Access Machine

The **Parallel Random Access Machine**, in short **PRAM** model, has p processing units that are all connected to a common *unbounded shared memory* (Fig. 2.2). Each processing unit can, in *one* step, access *any* location (word) in the shared memory by issuing a memory request *directly* to the shared memory.

The PRAM model of parallel computation is idealized in several respects. First, there is no limit on the number p of processing units, except that p is finite. Next, also idealistic is the assumption that a processing unit can access any location in the shared memory in one single step. Finally, for words in the shared memory it is only assumed that they are of the same size; otherwise they can be of arbitrary finite size.

Note that in this model there is no interconnection network for transferring memory requests and data back and forth between processing units and shared memory. (This will radically change in the other two models, the LMM (see Sect. 2.3.2) and the MMM (see Sect. 2.3.3).)

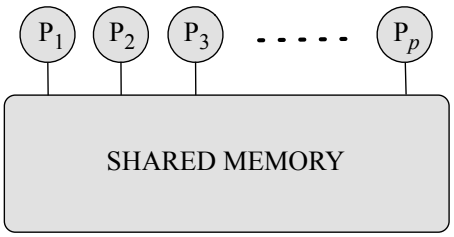


Fig. 2.2: The PRAM model of parallel computation: p processing units share an unbounded memory. Each processing unit can in one step access any memory location

However, the assumption that any processing unit can access any memory location in one step is **unrealistic**. To see why, suppose that processing units P_i and P_j simultaneously issue instructions I_i and I_j where both instructions intend to access (for reading from or writing to) the *same* memory location L (see Fig. 2.3).

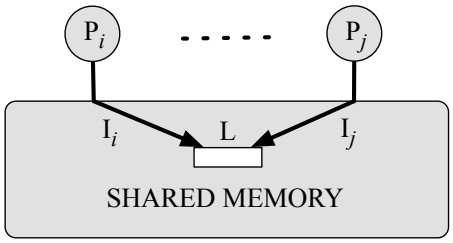


Fig. 2.3: Hazards of simultaneous access to a location. Two processing units simultaneously issue instructions each of which needs to access the same location L

Even if a truly simultaneous physical access to L had been possible, such an access could have resulted in unpredictable contents of L. Imagine what would be the contents of L after simultaneously writing 3 and 5 into it. Thus, it is reasonable to assume that, eventually, actual accesses of I_i and I_j to L are somehow, on the fly serialized (sequentialized) by hardware so that I_i and I_j physically access L *one after the other*.

Does such an implicit serialization neutralize all hazards of simultaneous access to the same location? Unfortunately not so. The reason is that the order of physical accesses of I_i and I_j to L is *unpredictable*: after the serialization, we cannot know whether I_i will physically access L *before* or *after* I_j.

Consequently, also the effects of instructions I_i and I_j are unpredictable (Fig. 2.3). Why? If *both* P_i and P_j want to *read* simultaneously from L, the instructions I_i and I_j will both read the same contents of L, regardless of their serialization, so both processing units will receive the same contents of L—as expected. However, if *one* of the processing units wants to *read* from L and the *other* simultaneously wants to *write* to L, then the data received by the reading processing unit will depend on whether the reading instruction has been serialized before or after the writing instruction. Moreover, if both P_i and P_j simultaneously attempt to *write* to L, the resulting contents of L will depend on how I_i and I_j have been serialized, i.e., which of I_i and I_j was the last to physically write to L.

In sum, simultaneous access to the same location may end in unpredictable data in the accessing processing units as well as in the accessed location.

In view of these findings it is natural to ask: Does this unpredictability make the PRAM model useless? The answer is no, as we will see shortly.

The variants of PRAM

The above issues led researchers to define several variations of PRAM that differ in

i) which sorts of simultaneous accesses to the same location are allowed; and

ii) the way in which unpredictability is avoided when simultaneously accessing the same location.

The variations are called the

- *Exclusive Read Exclusive Write PRAM* (*EREW-PRAM*),
- *Concurrent Read Exclusive Write PRAM* (*CREW-PRAM*), and
- *Concurrent Read Concurrent Write PRAM* (*CRCW-PRAM*).

We now describe them into more detail:

- **EREW**-PRAM. This is the most realistic of the three variations of the PRAM model. The EREW-PRAM model does not support simultaneous accessing to the same memory location; if such an attempt is made, the model stops executing its program. Accordingly, the implicit assumption is that programs running on EREW-PRAM never issue instructions that would simultaneously access the same location; that is, any access to any memory location must be *exclusive*. So the construction of such programs is the responsibility of algorithm designers.
- **CREW**-PRAM. This model supports simultaneous reads from the same memory location but requires exclusive writes to it. Again, the burden of constructing such programs is on the algorithm designer.
- **CRCW**-PRAM. This is the least realistic of the three versions of the PRAM model. The CRCW-PRAM model allows simultaneous reads from the same memory location, simultaneous writes to the same memory location, and simultaneous reads from and writes to the same memory location. However, to avoid unpredictable effects, different additional restrictions are imposed on simultaneous writes. This yields the following versions of the model CRCW-PRAM:

  – **CONSISTENT**-CRCW-PRAM. Processing units may simultaneously attempt to write to L, but it is assumed that they *all need to write the same value* to L. To guarantee that is, of course, the responsibility of the algorithm designer.

  – **ARBITRARY**-CRCW-PRAM. Processing units may simultaneously attempt to write to L (not necessarily the same value), but it is assumed that *only one of them will succeed*. Which processing unit will succeed is not predictable, so the programmer must take this into account when designing the algorithm.

  – **PRIORITY**-CRCW-PRAM. There is a priority order imposed on the processing units; e.g., the processing unit with smaller index has higher priority. Processing units may simultaneously attempt to write to L, but it is assumed that only the one with *the highest priority will succeed*. Again, algorithm designer must foresee and mind every possible situation during the execution.

– **FUSION**-CRCW-PRAM. Processing units may simultaneously attempt to write to L, but it is assumed that
  ◇ first a particular operation, denoted by $\circ$, will be applied on-the-fly to all the values $v_1, v_2, \ldots, v_k$ to be written to L, and
  ◇ only then the result $v_1 \circ v_2 \circ \cdots \circ v_k$ of the operation $\circ$ will be written to L.

  The operation $\circ$ is assumed to be associative and commutative, so that the value of the expression $v_1 \circ v_2 \circ \cdots \circ v_k$ does not dependent on the order of performing the operations $\circ$. Examples of the operation $\circ$ are the sum (+), product ($\cdot$), maximum (max), minimum (min), logical conjunction ($\wedge$), and logical disjunction ($\vee$).

$\star$ The relative power of the variants

As the restrictions of simultaneous access to the same location are relaxed when we pass from EREW-PRAM to CREW-PRAM and then to CRCW-PRAM, the variants of PRAM are becoming less and less realistic. On the other hand, as the restrictions are dropped, it is natural to expect that the variants may be gaining in their power. So we pose the following question:

Do EREW-PRAM, CREW-PRAM and CRCW-PRAM differ in their power?

The answer is *yes, but not too much.* The foggy "too much" is clarified in the next Theorem, where CRCW-PRAM(p) denotes the CRCW-PRAM with p processing units, and similarly for the EREW-PRAM(p). Informally, the theorem tells us that by passing from the EREW-PRAM(p) to the "more powerful' CRCW-PRAM(p) the parallel execution time of a parallel algorithm *may* reduce by some factor; however, this factor is *bounded above* and, indeed, it is at most of the order $O(\log p)$.

Theorem 2.1. *Every algorithm for solving a computational problem Π on the CRCW-PRAM(p) is at most $O(\log p)$-times faster than the fastest algorithm for solving Π on the EREW-PRAM(p).*

Proof Idea. We first show that CONSISTENT-CRCW-PRAM(p)'s simultaneous writings to the same location can be performed by EREW-PRAM(p) in $O(\log p)$ steps. Consequently, EREW-PRAM can simulate CRCW-PRAM, with slowdown factor $O(\log p)$. Then we show that this slowdown factor is tight, that is, there *exists* a computational problem Π for which the slowdown factor is actually $\Theta(\log p)$. Such a Π is, for example, the problem of *finding the maximum of n numbers*. □

Relevance of the PRAM model

We have explained why the PRAM model is unrealistic in the assumption of an immediately addressable, unbounded shared memory. Does this necessarily mean that the PRAM model is *irrelevant* for the purposes of practical implementation of parallel computation? The answer depends on *what we expect from the PRAM model* or, more generally, *how we understand the role of theory.*

When we strive to design an algorithm for solving a problem Π on PRAM, our efforts may not end up with a *practical* algorithm, ready for solving Π. However, the design may reveal something inherent to Π, namely, that Π is *parallelizable*. In other words, the design may detect in Π subproblems some of which could, at least in principle, be solved in parallel. In this case it usually proves that such subproblems are indeed *solvable in parallel* on the most liberal (and unrealistic) PRAM, the CRCW-PRAM.

At this point the importance of Theorem 2.1 becomes apparent: we can replace CRCW-PRAM by the *realistic* EREW-PRAM and solve Π on the latter. (All of that at the cost of a limited degradation in the speed of solving Π.)

In sum, the relevance of PRAM is reflected in the following method:

1. *Design* a program P for solving Π on the model CRCW-PRAM(p), where p may depend on the problem Π. Note that the design of P for CRCW-PRAM is expected to be easier than the design for EREW-PRAM, simply because CRCW-PRAM has no simultaneous-access restrictions to be taken into account.
2. *Run* P on EREW-PRAM(p), which is assumed to be able to simulate simultaneous accesses to the same location.
3. *Use* Theorem 2.1 to guarantee that the parallel execution time of P on EREW-PRAM(p) is at most $O(\log p)$-times higher than it would be on the less realistic CRCW-PRAM(p).

2.3.2 The Local-Memory Machine

The **LMM model** has p processing units, each with its own **local memory** (Fig. 2.4). The processing units are connected to a common **interconnection network**. Each processing unit can access its own local memory *directly*. In contrast, it can access a *non*-local memory (i.e., local memory of another processing unit) only by sending a **memory request** through the interconnection network.

The assumption is that all *local operations*, including accessing the local memory, take *unit time*. In contrast, the time required to access a non-local memory depends on

- the capability of the interconnection network and
- the pattern of coincident non-local memory accesses of other processing units as the accesses may congest the interconnection network.

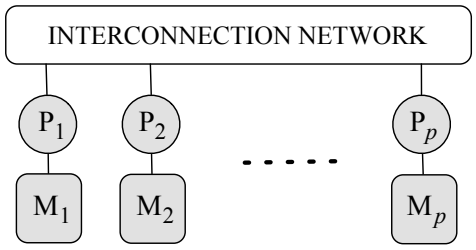


Fig. 2.4: The LMM model of parallel computation has p processing units each with its local memory. Each processing unit directly accesses its local memory and can access other processing unit's local memory via the interconnection network.

2.3.3 The Memory-Module Machine

The **MMM model** (Fig. 2.5) consists of p processing units and m **memory modules** each of which can be accessed by any processing unit via a *common interconnection network*. There are no local memories to processing units. A processing unit can access the memory module by sending a *memory request* through the interconnection network.

It is assumed that the processing units and memory modules are arranged in such a way that—when there are no coincident accesses—the time for any processing unit to access any memory module is roughly uniform. However, when there are coincident accesses, the access time depends on

- the capability of the interconnection network and
- the pattern of coincident memory accesses.

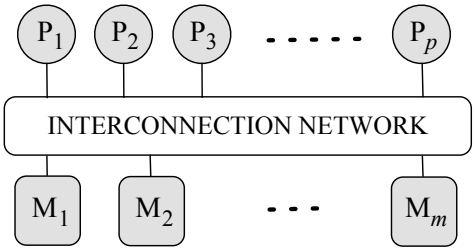


Fig. 2.5: The MMM model of parallel computation has p processing units and m memory modules. Each processing unit can access any memory module via the interconnection network. There are no local memories to the processing units.

2.4 The impact of communication

We have seen that both LMM model and MMM model explicitly use interconnection networks to convey memory requests to the non-local memories (see Fig. 2.4 and Fig. 2.5). In this section we focus on the role of an interconnection network in a multiprocessor model and its impact on the the parallel time complexity of parallel algorithms.

2.4.1 Interconnection networks

Since the dawn of parallel computing, the major hallmark of a parallel system have been the type of *the central processing unit* (CPU) and the *interconnection network*. This is now changing. Recent experiments have shown that execution times of most real world parallel applications are becoming more and more dependent on the *communication time* rather than on the calculation time. So, as the number of cooperating processing units or computers increases, the performance of interconnection networks is becoming more important than the performance of the processing unit. Specifically, the interconnection network has great impact on the *efficiency* and *scalability* of a parallel computer on most real world parallel applications. In other words, high performance of an interconnection network may ultimately reflect in higher speedups, because such an interconnection network can shorten the overall parallel execution time as well as increase the number of processing units that can be efficiently exploited.

The performance of an interconnection network depends on several factors. Three of the most important are the *routing*, the *flow-control algorithms*, and the *network topology*. Here **routing** is the process of selecting a path for traffic in an interconnection network; **flow control** is the process of managing the rate of data transmission between two nodes to prevent a fast sender from overwhelming a slow receiver; and **network topology** is the arrangement of the various elements, such as communication nodes and channels, of an interconnection network.

For the routing and flow-control algorithms efficient techniques are already known and used. In contrast, network topologies haven't been adjusting to changes in technological trends as promptly as the routing and flow-control algorithms. This is one reason that many network topologies which were discovered soon after the very birth of parallel computing are still being widely used. Another reason is the freedom that end users have when they are choosing the appropriate network topology for the anticipated usage. (Due to modern standards, there is no such freedom in picking or altering routing or flow-control algorithms.) As a consequence, a further step in performance increase can be expected to come from the improvements in the topology of interconnection networks. For example, such improvements should enable interconnection networks to dynamically adapt to the current application in some optimal way.

2.4.2 Basic properties of interconnection networks

We can classify interconnection networks in many ways and characterize them by various parameters. For defining most of these parameters, *graph theory* is the most elegant mathematical framework. More specifically, an interconnection network can be modeled as a **graph** $G(N,C)$, where N is a set of **communication nodes** and C is a set of **communication links** (or, **channels**) between the communication nodes. Based on this graph-theoretical view of interconnection networks, we can define parameters that represent both **topological properties** and **performance properties** of interconnection networks. Let us describe both kinds of properties.

Topological properties of interconnection networks

The most important topological properties of interconnection networks, defined by graph-theoretical notions, are the

- node degree,
- regularity,
- symmetry,
- diameter,
- path diversity, and
- expansion scalability.

In the following we define each of them and give comments where appropriate:

- The **node degree** is the number d of channels through which a communication node is connected to other communication nodes. Notice, that node degree includes only the *ports for the network communication*, although a communication node also needs ports for the connection to the processing element(s) and ports for service or maintenance channels.
- An interconnection network is said to be **regular** if all communication nodes have the same node degree; that is, there is a $d > 0$ such that every communication node has node degree d.
- An interconnection network is said to be **symmetric** if all communication nodes possess the "same view" of the network; that is, there is a homomorphism that maps any communication node to any other communication node. In a symmetric interconnection network, the load can be evenly distributed through all communication nodes, thus reducing congestion problems. Many real implementations of interconnection networks are based on *symmetric regular* graphs because of their fruitful topological properties that lead to a simple routing and fair load balancing under the uniform traffic.
- In order to move from a source node to a destination node, a packet must traverse through a series of elements, such as routers or switches, that together comprise a *path* (or, *route*) between the source and the destination node. The number of communication nodes traversed by the packet along this path is called the **hop**

count. In the best case, two nodes communicate through the path which has the **minimum hop count**, l, taken over all paths between the two nodes. Since l may vary with the source and destination nodes, we also use the **average distance**, l_{avg}, which is average l taken over all possible pairs of nodes. An important characteristic of any topology is the **diameter**, l_{max}, which is the maximum of all the minimum hop counts, taken over all pairs of source and destination nodes.

- In an interconnection network, there may exist multiple paths between two nodes. In such case, the nodes can be connected in many ways. A packet starting at source node will have at its disposal multiple routes to reach the destination node. The packet can take different routes (or even different continuations of a traversed part of a route) depending on the current situation in the network. An interconnection network that has high **path diversity** offers more alternatives when packets need to seek their destinations and/or avoid obstacles.
- **Scalability** is (i) the capability of a system to handle a growing amount of work, or (ii) the potential of the system to be enlarged to accommodate that growth. The scalability is important at every level. For example, the basic building block must be easily connected to other blocks in a uniform way. Moreover, the same building block must be used to build interconnection networks of different sizes, with only a small performance degradation for the maximum-size parallel computer. Interconnection networks have important impact on scalability of parallel computers that are based on the LMM or MMM multiprocessor model. To appreciate that, note that scalability is limited if node degree is fixed.

Performance properties of interconnection networks

The main performance properties of interconnection networks are the

- channel bandwidth,
- bisection bandwidth, and
- latency.

We now define each of them and give comments where appropriate:

- **Channel bandwidth**, in short bandwidth, is the amount of data that is, or theoretically could be, communicated through a channel in a given amount of time. In most cases, the channel bandwidth can be adequately determined by using a simple **model of communication** which advocates that the **communication time** t_{comm}, needed to communicate given data through the channel, is the sum $t_{\text{s}} + t_{\text{d}}$ of the **start-up time** t_{s}, needed to set-up the channel's software and hardware, and the **data transfer time** t_{d}, where $t_{\text{d}} = mt_{\text{w}}$, the product of the number of words making up the data, m, and the **transfer time per one word**, t_{w}. Then the channel bandwidth is $1/t_{\text{w}}$.
- A given interconnection network can be *cut* into two (almost) equal-sized components. Generally, this can be done in many ways. Given a cut of the interconnection network, the *cut-bandwidth* is the sum of channel bandwidths of all chan-

nels connecting the two components. The smallest cut-bandwidth is called the **bisection bandwidth** (BBW) of the interconnection network. The corresponding cut is the *worst-case cut* of the interconnection network. Occasionally, the **bisection bandwidth per node** (BBWN) is needed; we define it as BBW divided by $|N|$, the number of nodes in the network. Of course, both BBW and BBWN depend on the topology of the network and the channel bandwidths. All in all, increasing the bandwidth of the interconnection network can have as beneficial effects as increasing the CPU clock (recall Sect. 2.4.1).

- **Latency** is the time required for a packet to travel from the source node to the destination node. Many applications, especially those using short messages, are latency sensitive in the sense that efficiencies of these applications strongly depend on the latency. For such applications, their software overhead may become a major factor that influences the latency. Ultimately, the latency is bounded below by the time in which light traverses the physical distance between two nodes.

The transfer of data from a source node to a destination node is measured in terms of various units which are defined as follows:

- **packet**, the smallest amount of data that can be transferred by hardware,
- **FLIT** (flow control digit), the amount of data used to allocate the buffer space in some flow-control techniques;
- **PHIT** (physical digit), the amount of data that can be transferred in a single cycle.

These units are closely related to the bandwidth and to the latency of the network.

Mapping interconnection networks into real space

An interconnection network of any given topology, even if defined in an abstract higher-dimensional space, eventually has to be **mapped** into the physical, three-dimensional (3D) space. This means that all the chips and printed-circuit boards making up the interconnection network must be allocated physical places.

Unfortunately, this is not a trivial task. The reason is that mapping usually has to optimize certain, often contradicting, criteria while at the same time respecting various restrictions. Here are some examples:

- One such restriction is that the numbers of I/O pins per chip or per printed-circuit board are bounded above. A usual optimization criterion is that, in order to prevent the decrease of data rate, cables be as short as possible. But due to significant sizes of hardware components and due to physical limitations of 3D-space, mapping may considerably stretch certain paths, i.e., nodes that are close in higher-dimensional space may be mapped to distant locations in 3D-space.
- We may want to map processing units that communicate intensively as close together as possibly, ideally on the same chip. In this way we may minimize the impact of communication. Unfortunately, the construction of such optimal mappings is NP-hard optimization problem.
- An additional criterion may be that the power consumption is minimized.

2.4.3 Classification of interconnection networks

Interconnection networks can be classified into *direct* and *indirect* networks. Here are the main properties of each kind.

Direct networks

A network is said to be **direct** when each node is *directly connected* to its neighbors. How many neighbors can a node have? In a **fully connected network**, each of the $n = |N|$ nodes is directly connected to *all* the other nodes, so each node has $n-1$ neighbors. (See Fig. 2.6.)

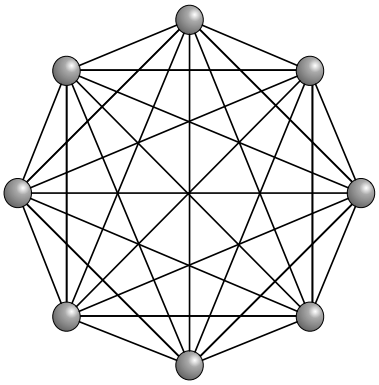


Fig. 2.6: A fully connected network with $n = 8$ nodes.

Since such a network has $\frac{1}{2}n(n-1) = \Theta(n^2)$ direct connections, it can only be used for building systems with small numbers n of nodes. When n is large, each node is directly connected to a *proper subset* of other nodes, while the communication to the remaining nodes is achieved by routing messages through intermediate nodes. An example of such a direct interconnection network is the hypercube; see Fig. 2.13 on p. 29.

Indirect networks

An **indirect** network connects the nodes through **switches**. Usually, it connects processing units on one end of the network and memory modules on the other end of the network. The simplest circuit for connecting processing units to memory modules is the **fully connected crossbar switch** (Fig. 2.7). Its advantage is that it can establish a connection between processing units and memory modules in an arbitrary way.

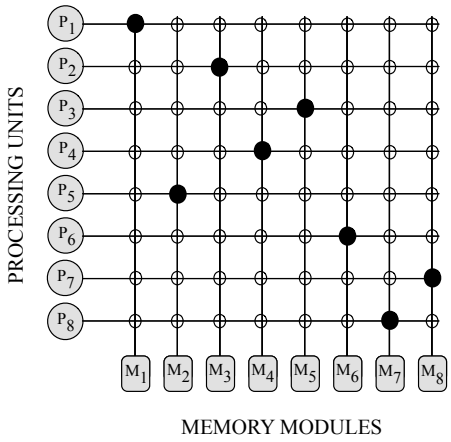


Fig. 2.7: A fully connected crossbar switch connecting 8 nodes to 8 nodes.

At each intersection of a horizontal and vertical line is a crosspoint. A crosspoint is a small switch that can be electrically opened (○) or closed (●), depending on whether the horizontal and vertical lines are to be connected or not. In Fig. 2.7 we see eight crosspoints closed simultaneously, allowing connections between the pairs (P_1, M_1), (P_2, M_3), (P_3, M_5), (P_4, M_4), (P_5, M_2), (P_6, M_6), (P_7, M_8) and (P_8, M_7) at the same time. Many other combinations are also possible.

Unfortunately, the fully connected crossbar has too large complexity to be used for connecting large numbers of input and output ports. Specifically, the number of crosspoints grows as pm, where p and m are the numbers of processing units and memory modules, respectively. For $p = m = 1000$ this amounts to a million crosspoints which is not feasible. (Nevertheless, for medium-sized systems, a crossbar design is workable, and small fully connected crossbar switches are used as basic building blocks within larger switches and routers.)

This is why indirect networks connect the nodes through many switches. The switches themselves are usually connected to each other in **stages**, using a regular connection pattern between the stages. Such indirect networks are called the *multistage* interconnection networks; we will describe them in more detail on p. 30.

Indirect networks can be further classified as follows:

- A **non-blocking** network can connect any idle source to any idle destination, regardless of the connections already established across the network. This is due to the network topology which ensures the existence of multiple paths between the source and destination.
- A **blocking rearrangeable** networks can rearrange the connections that have already been established across the network in such a way that a new connection can be established. Such a network can establish all possible connections between inputs and outputs.

- In a **blocking** network, a connection that has been established across the network may block the establishment of a new connection between a source and destination, even if the source and destination are both free. Such a network cannot always provide a connection between a source and an arbitrary free destination.

The distinction between direct and indirect networks is less clear nowadays. Every direct network can be represented as an indirect network since every node in the direct network can be represented as a router with its own processing element connected to other routers. However, for both direct and indirect interconnection networks, the full crossbar, as an ideal switch, is the heart of the communications.

2.4.4 Topologies of interconnection networks

It is not hard to see that there exist many network topologies capable of interconnecting p processing units and m memory modules (see Exercises). However, not every network topology is capable of conveying memory requests quickly enough to *efficiently* back up parallel computation. Moreover, it turns out that the network topology has a large influence on the *performance* of the interconnection network and, consequently, of parallel computation. In addition, network topology may incur considerable difficulties in the actual construction of the network and its cost.

In the last few decades, researchers have proposed, analyzed, constructed, tested, and used various network topologies. We now give an overview of the most notable or popular ones: the *bus*, the *mesh*, the *3D-mesh*, the *torus*, the *hypercube*, the *multistage network* and the *fat tree*.

The bus

This is the simplest network topology. See Fig. 2.8. It can be used in both local-memory machines (LMMs) and memory-module machines (MMMs). In either case, all processing units and memory modules are connected to a single bus. In each step, at most one piece of data can be written onto the bus. This can be a request from a processing unit to read or write a memory value, or it can be the response from the processing unit or memory module that holds the value.

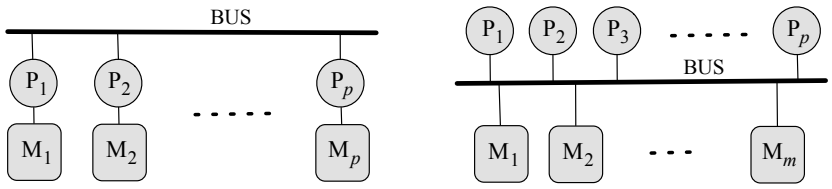


Fig. 2.8: The bus is the simplest network topology.

When in a memory-module machine a processing unit wants to read a memory word, it must first check to see if the bus is busy. If the bus is idle, the processing unit puts the address of the desired word on the bus, issues the necessary control signals, and waits until the memory puts the desired word on the bus. If, however, the bus is busy when a processing unit wants to read or write memory, the processing unit must *wait until the bus becomes idle*. This is where drawbacks of the bus topology become apparent. If there is a small number of processing units, say two or three, the contention for the bus is manageable; but for larger numbers of processing units, say 32, the contention becomes unbearable because most of the processing units will wait most of the time.

To solve this problem we add a *local cache to each processing unit*. The cache can be located on the processing unit board, next to the processing unit chip, inside the processing unit chip, or some combination of all three. In general, caching is not done on an individual word basis but on the basis of blocks that consist of, say, 64 bytes. When a word is referenced by a processing unit, the word's entire block is fetched into the local cache of the processing unit. After that many reads can be satisfied out of the local cache. As a result, there will be less bus traffic, and the system will be able to support more processing units.

We see, that the practical advantages of using buses are that (i) they are simple to build, and (ii) it is relatively easy to develop protocols that allow processing units to cache memory values locally (because all processing units and memory modules can observe the traffic on the bus). The obvious disadvantage of using a bus is that the processing units must take turns accessing the bus. This implies that as more processing units are added to a bus, the average time to perform a memory access grows proportionately with the number of processing units.

The ring

The ring is among the simplest and the oldest interconnection networks. Given n nodes, they are arranged in linear fashion so that each node has a distinct label i, where $0 \leqslant i \leqslant n-1$. Every node is connected to two neighbors, one to the left and one to the right. Thus, a node labeled i is connected to the nodes labeled $i+1 \bmod n$ and $i-1 \bmod n$ (see Fig. 2.9). The ring is used in local-memory machines (LMMs).

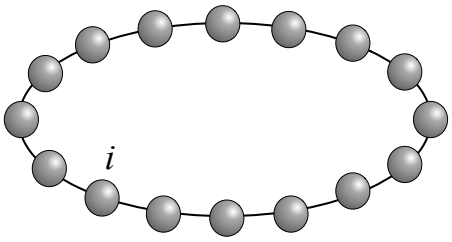


Fig. 2.9: A ring. Each node represents a processor unit with local memory

2D-mesh

A two-dimensional mesh is an interconnection network that can be arranged in rectangular fashion, so that each switch in the mesh has a distinct label (i, j), where $0 \leqslant i \leqslant X-1$ and $0 \leqslant j \leqslant Y-1$. (See Fig. 2.10.) The values X and Y determine the lengths of the sides of the mesh. Thus, the number of switches in a mesh is XY. Every switch, except those on the sides of the mesh, is connected to six neighbors: one to the north, one to the south, one to the east, and one to the west. So a switch labeled (i, j), where $0 < i < X-1$ and $0 < j < Y-1$, is connected to the switches labeled $(i, j+1)$, $(i, j-1)$, $(i+1, j)$, and $(i-1, j)$.

Meshes typically appear in local-memory machines (LMMs): a processing unit (along with its local memory) is connected to each switch, so that remote memory accesses are made by routing messages through the mesh.

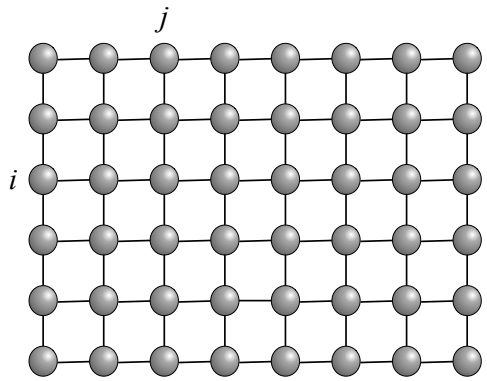


Fig. 2.10: A 2D-mesh. Each node represents a processor unit with local memory

2D-torus (toroidal 2D-mesh)

In the 2D-mesh, the switches on the sides have no connections to the switches on the opposite sides. The interconnection network that compensates for this is called the toroidal mesh, or just torus when $d = 2$. (See Fig. 2.11.) Thus, in torus every switch located at (i, j) is connected to four other switches, which are located at $(i, j+1 \bmod Y)$, $(i, j-1 \bmod Y)$, $(i+1 \bmod X, j)$ and $(i-1 \bmod X, j)$.

Toruses appear in local-memory machines (LMMs): to each switch is connected a processing unit with its local memory. Each processing unit can access any remote memory by routing messages through the torus.

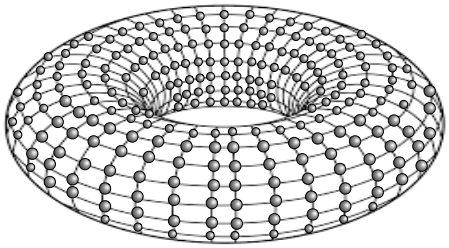


Fig. 2.11: A 2D-torus. Each node represents a processor unit with local memory

3D-mesh and 3D-torus

A three-dimensional mesh is similar to two-dimensional. (See Fig. 2.12.) Now each switch in a mesh has a distinct label (i, j, k), where $0 \leqslant i \leqslant X-1$, $0 \leqslant j \leqslant Y-1$, and $0 \leqslant k \leqslant Z-1$. The values X, Y and Z determine the lengths of the sides of the mesh, so the number of switches in it is XYZ. Every switch, except those on the sides of the mesh, is now connected to six neighbors: one to the north, one to the south, one to the east, one to the west, one up, and one down. Thus, a switch labeled (i, j, k), where $0 < i < X-1$, $0 < j < Y-1$ and $0 < k < Z-1$, is connected to the switches $(i, j+1, k)$, $(i, j-1, k)$, $(i+1, j, k)$, $(i-1, j, k)$, $(i, j, k+1)$ and $(i, j, k-1)$. Such meshes typically appear in LMMs.

We can expand a 3D-mesh into a **toroidal 3D-mesh** by adding edges that connect nodes located at the opposite sides of the 3D-mesh. (Picture omitted.) A switch labeled (i, j, k) is connected to the switches $(i+1 \bmod X, j, k)$, $(i-1 \bmod X, j, k)$, $(i, j+1 \bmod Y, k)$, $(i, j-1 \bmod Y, k)$, $(i, j, k+1 \bmod Z)$ and $(i, j, k-1 \bmod Z)$.

3D-meshes and toroidal 3D-meshes are used in local-memory machines (LMMs).

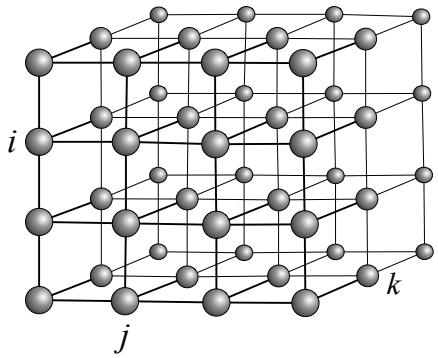


Fig. 2.12: A 3D-mesh. Each node represents a processor unit with local memory

Hypercube

A hypercube is an interconnection network that has $n = 2^b$ nodes, for some $b \geqslant 0$. (See Fig. 2.13.) Each node has a distinct label consisting of b bits. Two nodes are connected by a communication link if an only if their labels differ in precisely one bit location. Hence, each node of a hypercube has $b = \log_2 n$ neighbors.

Hypercubes are used in local-memory machines (LMMs).

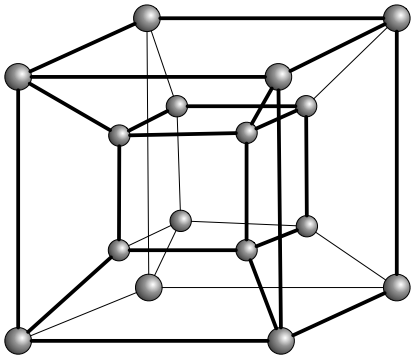


Fig. 2.13: A hypercube. Each node represents a processor unit with local memory

⋆ The k-ary d-cube family of network topologies

Interestingly, the ring, the 2D-torus, the 3D-torus, the hypercube, and many other topologies all belong to one larger family of **k-ary d-cube** topologies.

Given $k \geqslant 1$ and $d \geqslant 1$, the k-ary d-cube topology is a *family* of certain "gridlike" topologies that share the fashion in which they are constructed. In other words, the k-ary d-cube topology is a *generalization* of certain topologies. The parameter d is called the **dimension** of these topologies and k is their **side length**, the number of nodes along each of the d directions. The fashion in which the k-ary d-cube topology is constructed is defined *inductively* (on the dimension d):

A k-ary d-cube is constructed from k other k-ary $(d-1)$-cubes
by connecting the nodes with identical positions into rings.

This inductive definition enables us to systematically construct actual k-ary d-cube topologies and analyze their topological and performance properties. For instance, we can deduce that a k-ary d-cube topology contains $n = k^d$ communication nodes and $c = dn = dk^d$ communication links, while the diameter is $l_{max} = \frac{dk}{2}$ and the average distance between two nodes is $l_{avg} = \frac{l_{max}}{2}$ (if k even) or $l_{avg} = d(\frac{k}{4} - \frac{1}{4k})$ (if k odd). Unfortunately, in spite of their simple recursive structure, the k-ary d-cubes have a poor expansion scalability.

Multistage network

A multistage network connects one set of switches, called the *input switches*, to another set, called the *output switches*. The network achieves this through a sequence of *stages*, where each stage consists of switches. (See Fig. 2.14.) In particular, the input switches form the first stage, and the output switches form the last stage. The number d of stages is called the *depth* of the multistage network. Usually, a multistage network allows to send a peace of data from any input switch to any output switch. This is done along a path that traverses all the stages of the network in order from 1 to d. There are many different multistage network topologies.

Multistage networks are frequently used in memory-module machines (MMMs); there, processing units are attached to input switches, and memory modules are attached to output switches.

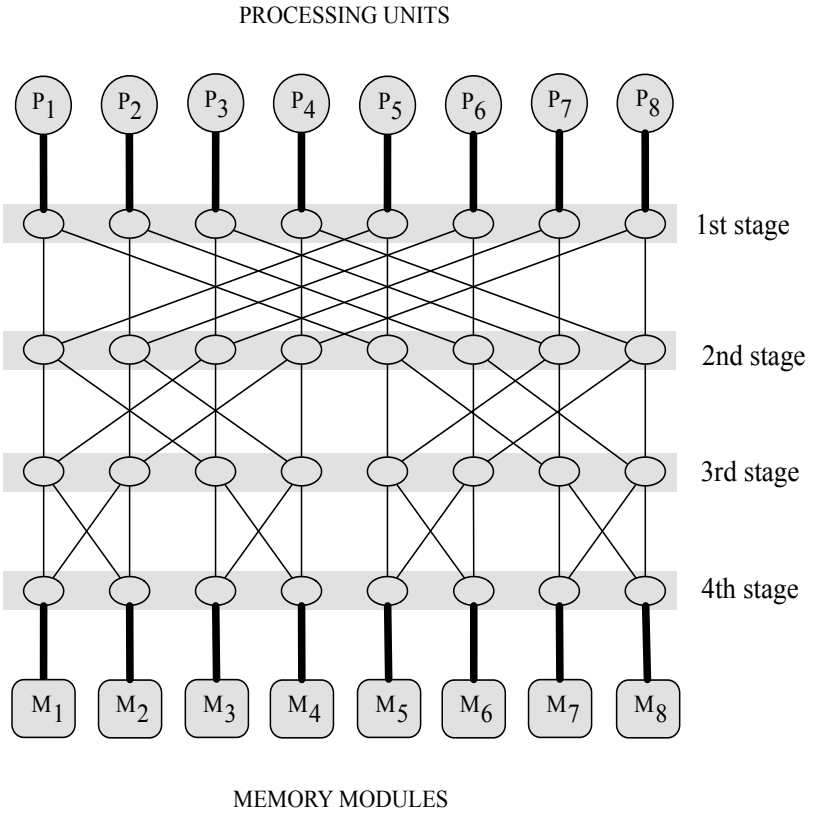


Fig. 2.14: A 4-stage interconnection network capable of connecting 8 processing units to 8 memory modules. Each switch ○ can establish a connection between arbitrary pair of input and output channels.

Fat tree

A fat tree is a network whose structure is based on that of a tree. (See Fig. 2.15.) However, in contrast to the usual tree where edges have the same thickness, in a fat tree, edges that are nearer the root of the tree are "fatter" (thicker) than edges that are further down the tree. The idea is that each node of a fat tree may represent many network switches, and each edge may represent many communication channels. The more channels an edge represents, the larger is its capacity and the fatter is the edge. So the capacities of the edges near the root of the fat tree are much larger than the capacities of the edges near the leaves.

Fat trees can be used to construct local-memory machines (LMMs): processing units along with their local memories are connected to the leaves of the fat tree, so that a message from one processing unit to another first travels up the tree to the least common ancestor of the two processing units and then down the tree to the destination processing unit.

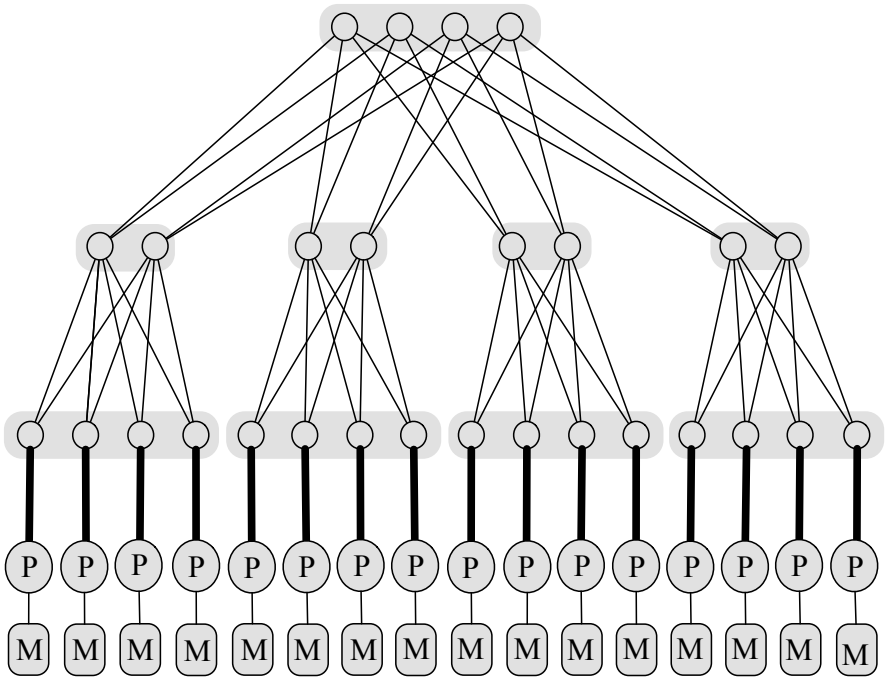


Fig. 2.15: A fat-tree. Each switch ○ can establish a connection between arbitrary pair of incident channels.

2.5 Parallel computational complexity

In order to examine the complexity of computational problems and their parallel algorithms, we need some new basic notions. We will now introduce a few of these.

2.5.1 Problem instances and their sizes

Let Π be a computational problem. In practice we are usually confronted with a particular *instance* of the problem Π. The instance is obtained from Π by replacing the variables in the definition of Π with actual data. Since this can be done in many ways, each way resulting in a different instance of Π, we see that the problem Π can be viewed as a *set* of all the possible instances of Π.

To each instance π of Π we can associate a natural number which we call the **size of the instance** π and denote by

$$\text{size}(\pi).$$

Informally, size(π) is roughly the amount of space needed to represent π in some way accessible to a computer. and, in practice, depends on the problem Π.

For example, if we choose $\Pi \equiv$ "sort a given finite sequence of numbers," then $\pi \equiv$ "sort 0 9 2 7 4 5 6 3" is an instance of Π and size$(\pi) = 8$, the number of numbers to be sorted. If, however, $\Pi \equiv$ "Is the n a prime number?", then "Is the 17 a prime number?" is an instance π of Π with size$(\pi) = 5$, the numebr of bits in the binary representation of 17. And if Π is a problem about graphs, then the size of an instance of Π is often defined as the number of nodes in the actual graph.

Why do we need sizes of instances? When we examine how fast an algorithm A for a problem Π is, we usually want to know how A's execution time depends on the size of instances of Π that are input to A. More precisely, we want to find a function

$$T(n)$$

whose value at n will represent the execution time of A on instances of size n. As a matter of fact, we are mostly interested in the **rate of growth** of $T(n)$, that is, *how quickly $T(n)$ grows when n grows*.

For example, if we find that $T(n) = n$, then A's execution time is a **linear** function of n, so if we double the size of problem instances, A's execution time doubles too. More generally, if we find that $T(n) = n^{const}$ ($const \geqslant 1$), then A's execution time is a **polynomial** function of n; if we now double the size of problem instances, then A's execution time multiplies by 2^{const}. If, however, we find that $T(n) = 2^n$, which is an **exponential** function of n, then things become dramatic: doubling the size n of problem instances causes A to run 2^n-times longer! So, doubling the size from 10 to 20 and then to 40 and 80, the execution time of A increases 2^{10} ($\approx$ thousand) times, then 2^{20} ($\approx$ million) times, and finally 2^{40} ($\approx$ thousand billion) times.

2.5.2 Number of processing units vs. size of problem instances

In Section 2.1, we defined the parallel execution time T_{par}, speedup S, and efficiency E of a parallel program P for solving a problem Π on a computer $C(p)$ with p processing units. Let us augment these definitions so that they will involve the size n of the instances of Π. As before, the program P for solving Π and the computer $C(p)$ are tacitly understood, so we omit the corresponding indexes to simplify the notation. We obtain the parallel execution time $T_{\text{par}}(n)$, speedup $S(n)$, and efficiency $E(n)$ of *solving Π's instances of size n*:

$$S(n) \stackrel{\text{def}}{=} \frac{T_{\text{seq}}(n)}{T_{\text{par}}(n)},$$

$$E(n) \stackrel{\text{def}}{=} \frac{S(n)}{p}.$$

So let us pick an arbitrary n and suppose that we are only interested in solving instances of Π whose size is n. Now, if there are *too few* processing units in $C(p)$, i.e., p is too small, the potential parallelism in the program P will not be fully exploited during the execution of P on $C(p)$, and this will reflect in low speedup $S(n)$ of P. Likewise, if $C(p)$ has *too many* processing units, i.e., p is too large, some of the processing units will be idling during the execution of the program P, and again this will reflect in low speedup of P. This raises the following question that obviously deserves further consideration:

How many processing units p should have $C(p)$,
so that, for all instances of Π of size n, the speedup of P will be maximal?

It is reasonable to expect that the answer will depend somehow on the type of C, that is, on the multiprocessor model (see Sect. 2.3) underlying the parallel computer C. Until we choose the multiprocessor model, we may not be able to obtain answers of practical value to the above question. Nevertheless, we *can* make some general observations that hold for any type of C. First observe that, in general, if we let n grow then p must grow too; otherwise, p would eventually become too small relative to n, thus making $C(p)$ incapable of fully exploiting the potential parallelism of P. Consequently, we may view p, the number of processing units that are needed to maximize speedup, to be some function of n, the size of the problem instance at hand. In addition, intuition and practice tell us that a larger instance of a problem requires at least as many processing units as required by a smaller one. In sum, we can set

$$p = f(n),$$

where $f : \mathbb{N} \to \mathbb{N}$ is some *nondecreasing* function, i.e., $f(n) \leqslant f(n+1)$, for all n.

Second, let us examine how quickly can $f(n)$ grow as n grows? Suppose that $f(n)$ grows **exponentially.** Well, researchers have proved that if there are exponentially many processing units in a parallel computer then this necessarily incurs long communication paths between some of them. Since some communicating processing units become exponentially distant from each other, the communication times between them increase correspondingly and, eventually, blemish the theoretically achievable speedup. The reason for all of that is essentially in our real, 3-dimensional space, because

- each processing unit and each communication link occupies some non-zero volume of space, and
- the diameter of the smallest sphere containing exponentially many processing units and communication links is also exponential.

In sum, exponential number of processing units is impractical and leads to theoretically tricky situations.

Suppose now that $f(n)$ grows **polynomially**, i.e., f is a polynomial function of n. Calculus tells us that if $\text{poly}(n)$ and $\exp(n)$ are a polynomial and an exponential function, respectively, then there is an $n' > 0$ so that $\text{poly}(n) < \exp(n)$ for all $n > n'$; that is, $\text{poly}(n)$ is eventually dominated by $\exp(n)$. In other words, we say that a polynomial function $\text{poly}(n)$ **asymptotically grows slower** than an exponential function $\exp(n)$. Note that $\text{poly}(n)$ and $\exp(n)$ are two *arbitrary* functions of n.

So we have $f(n) = \text{poly}(n)$ and consequently the number of processing units is

$$p = \text{poly}(n),$$

where poly(n) is a polynomial function of n. Here we tacitly discard polynomial functions of "unreasonably" large degrees, e.g. n^{100}. Indeed, we are hoping for much lower degrees, such as 2, 3, 4 or so, which will yield realistic and affordable numbers p of processing units.

In summary, we have obtained an answer to the question above which—because of the generality of C and Π, and due to restrictions imposed by nature and economy—falls short of our expectation. Nevertheless, the answer tells us that p must be some polynomial function (of a moderate degree) of n.

We will apply this to Theorem 2.1 (p. 16) right away in the next section.

2.5.3 The class NC of efficiently parallelizable problems

Let P be an algorithm for solving a problem Π on CRCW-PRAM(p). According to Theorem 2.1, the execution of P on EREW-PRAM(p) will be at most $O(\log p)$-times slower than on CRCW-PRAM(p). Let us use the observations from previous section and require that $p = \text{poly}(n)$. It follows that $\log p = \log \text{poly}(n) = O(\log n)$. To appreciate why, see Exercises in Sect. 2.7.

Combined with Theorem 2.1 this means that for $p = \text{poly}(n)$ the execution of P on EREW-PRAM(p) will be at most $O(\log n)$-times slower than on CRCW-PRAM(p).

But this also tells us that, when $p = \text{poly}(n)$, choosing a model from the models CRCW-PRAM(p), CREW-PRAM(p), and EREW-PRAM(p) to execute a program affects the execution time of the program by a factor of the order $O(\log n)$, where n is the size of the problem instances to be solved. In other words:

The execution time of a program does not vary too much as we choose the variant of PRAM that will execute it.

This motivates us to introduce a *class of computational problems* containing all the problems that have "fast" parallel algorithms requiring "reasonable" numbers of processing units. But what do "fast" and "reasonable" really mean? We have seen in previous section that the number of processing units is reasonable if it is *polynomial* in n. As for the meaning of "fast", a parallel algorithm is considered to be fast if its parallel execution time is *polylogarithmic* in n. That is fine, but what does now "polylogarithmic" mean? Here is the definition.

Definition 2.1. A function is **polylogarithmic** in n if it is polynomial in $\log n$, i.e., if it is $a_k(\log n)^k + a_{k-1}(\log n)^{k-1} + \cdots + a_1(\log n)^1 + a_0$, for some $k \geqslant 1$.

We usually write $\log^i n$ instead of $(\log n)^i$ to avoid clustering of parentheses. The sum $a_k \log^k n + a_{k-1}\log^{k-1} n + \cdots + a_0$ is asymptotically bounded above by $O(\log^k n)$. To see why, consider Exercises in Sect. 2.7.

We are ready to formally introduce the class of problems we are interested in.

Definition 2.2. Let **NC** be the class of computational problems solvable in polylogarithmic time on PRAM with polynomial number of processing units.

If a problem Π is in the class *NC*, then it is solvable in polylogarithmic parallel time with polynomially many processing units *regardless of the variant* of PRAM used to solve Π. In other words, the class NC is **robust**, insensitive to the variations of PRAM. How can we see that? If we replace one variant of PRAM with another, then by Theorem 2.1 Π's parallel execution time $O(\log^k n)$ can only increase by a factor $O(\log n)$ to $O(\log^{k+1} n)$ which is still polylogarithmic.

In sum, NC is the class of **efficiently parallelizable** computational problems.

Example 2.1. Suppose that we are given the problem $\Pi \equiv$ "add n given numbers." Then $\pi \equiv$ "add numbers 10, 20, 30, 40, 50, 60, 70, 80" is an instance of size$(\pi) = 8$ of the problem Π. Let us now focus on *all* instances of size 8, that is, instances of the form $\pi \equiv$ "add numbers $a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8$."

The fastest sequential algorithm for computing the sum $a_1 + a_2 + a_3 + a_4 + a_5 + a_6 + a_7 + a_8$ requires $T_{\text{seq}}(8) = 7$ steps, with each step adding the next number to the sum of the previous ones.

In parallel, however, the numbers $a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8$ can be summed in just $T_{\text{par}}(8) = 3$ parallel steps using $\frac{8}{2} = 4$ processing units which communicate in a tree-like pattern as depicted in Fig. 2.16. In the first step, $s = 1$, each processing unit adds two adjacent input numbers. In each next step, $s \geqslant 2$, two adjacent previous partial results are added to produce a new, combined partial result. This combining of partial results in a tree-like manner continues until $2^{s+1} > 8$. In the first step, $s = 1$, all of the four processing units are engaged in computation; in step $s = 2$, two processing units (P_3 and P_4) start idling; and in step $s = 3$, three processing units (P_2, P_3 and P_4) are idle.

In general, instances $\pi(n)$ of Π can be solved in parallel time $T_{\text{par}} = \lceil \log n \rceil = O(\log n)$ with $\lceil \frac{n}{2} \rceil = O(n)$ processing units communicating in similar tree-like patterns. Hence, $\Pi \in$ NC and the associated speedup is $S(n) = \frac{T_{\text{seq}}(n)}{T_{\text{par}}(n)} = O(\frac{n}{\log n})$. □

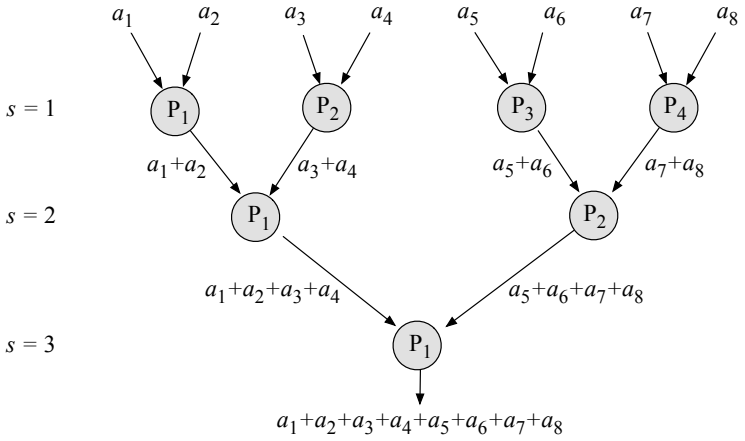


Fig. 2.16: Adding eight numbers in parallel with four processing units.

Notice that, in the above example, the efficiency of the tree-like parallel addition of n numbers is quite low, $E(n) = O(\frac{1}{\log n})$. The reason for this is obvious: only half of the processing units engaged in a parallel step s will be engaged in the next parallel step $s+1$, while all the other processing units will be idling until the end of computation. This issue will be addressed in the next section by Brent's Theorem.

2.6 Laws and theorems of parallel computation

In this section we describe the Brent's theorem, which is useful in estimating the lower bound on the number of processing units that are needed to keep a given parallel time complexity. Then we focus on the Amdahl's law, which is used for predicting the theoretical speedup of a parallel program whose different parts allow different speedups.

2.6.1 Brent's theorem

Brent's theorem enables us to quantify the performance of a parallel program when the number of processing units is reduced.

Let M be a PRAM of an arbitrary type and containing unspecified number of processing units. More specifically, we assume that the number of processing units is always sufficient to cover all the needs of any parallel program.

When a parallel program P is run on M, different numbers of operations of P are performed, at each step, by different processing units of M. Suppose that a total of

$$W$$

operations are performed during the parallel execution of P on M (W is also called the **work** of P), and denote the parallel runtime of P on M by

$$T_{\text{par, M}}(P).$$

Let us now reduce the number of processing units of M to some fixed number

$$p$$

and denote the obtained machine with the reduced number of processing units by

$$R.$$

R is a PRAM of the same type as M which can use, in every step of its operation, at most p processing units.

Let us now run P on R. If p processing units cannot support, in every step of the execution, all the potential parallelism of P, then the parallel runtime of P on R,

$$T_{\text{par, R}}(P),$$

may be larger than $T_{\text{par, M}}(P)$. Now the question raises: Can we quantify $T_{\text{par, R}}(P)$?

The answer is given by **Brent's Theorem** which states that

$$T_{\text{par, R}}(P) = O\left(\frac{W}{p} + T_{\text{par, M}}(P)\right).$$

Proof. Let W_i be the number of P's operations performed by M in ith step and $T := T_{\text{par, M}}(P)$. Then $\sum_{i=1}^{T} W_i = W$. To perform the W_i operations of the ith step of M, R needs $\lceil \frac{W_i}{p} \rceil$ steps. So the number of steps which R makes during its execution of P is $T_{\text{par, R}}(P) = \sum_{i=1}^{T} \lceil \frac{W_i}{p} \rceil \leqslant \sum_{i=1}^{T} \left(\frac{W_i}{p} + 1 \right) \leqslant \frac{1}{p} \sum_{i=1}^{T} W_i + T = \frac{W}{p} + T_{\text{par, M}}(P)$. □

Applications of Brent's theorem

Brent's Theorem is useful when we want to reduce the number of processing units as much as possible while keeping the parallel time complexity. For example, we have seen in Example 2.1 on p. 36 that we can sum up n numbers in parallel time $O(\log n)$ with $O(n)$ processing units. Can we do the same with asymptotically less processing units? Yes, we can. Brent's Theorem tells us that $O(n/\log n)$ processing units suffice to sum up n numbers in $O(\log n)$ parallel time. See Exercises in Sect. 2.7.

2.6.2 Amdahl's law

Intuitively, we would expect that *doubling* the number of processing units should *halve* the parallel execution time; and doubling the number of processing units again should halve the parallel execution time once more. In other words, we would expect that the speedup from parallelization is a linear function of the number of processing units (see Fig. 2.17).

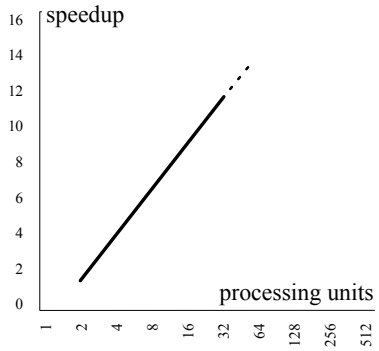


Fig. 2.17: Expected (linear) speedup as a function of the number of processing units.

However, linear speedup from parallelization is just a desirable optimum which is not very likely to become a reality. Indeed, in reality very few parallel algorithms achieve it. Most of parallel programs have a speedup which is *near-linear* for *small* numbers of processing elements, and then flattens out into a *constant* value for *large* numbers of processing elements (see Fig. 2.18).

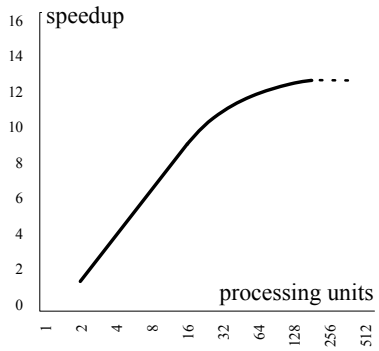


Fig. 2.18: Actual speedup as a function of the number of processing units.

Setting the stage

How can we explain this unexpected behavior? The clues for the answer will be obtained from two simple examples.

- *Example 1.* Let P be a sequential program processing files from disk as follows:
  - P is a sequence of two parts, $P = P_1P_2$;
  - P_1 scans the directory of the disk, creates a list of file names, and hands the list over to P_2;
  - P_2 passes each file from the list to the processing unit for further processing.

  Note: P_1 *cannot* be sped up by adding new processing units, because scanning the disk directory is intrinsically sequential process. In contrast, P_2 *can* be sped up by adding new processing units; for example, each file can be passed to a separate processing unit. In sum, *a sequential program can be viewed as a sequence of two parts that differ in their parallelizability*, i.e., amenability to parallelization.

- *Example 2.* Let P be as above. Suppose that the (sequential) execution of P takes 20 minutes, where the following holds (see Fig. 2.19):
  - the non-parallelizable P_1 runs *2 minutes*;
  - the parallelizable P_2 runs *18 minutes*.

  Note: since only P_2 can benefit from additional processing units, the parallel execution time $T_{\text{seq}}(P)$ of the whole P cannot be less than the time $T_{\text{seq}}(P_1)$ taken by the non-parallelizable part P_1 (that is, 2 minutes), regardless of the number of additional processing units engaged in the parallel execution of P. In sum, if parts of a sequential program differ in their potential parallelisms, they differ in their potential speedups from the increased number of processing units, so the speedup of the whole program will depend on their sequential runtimes.

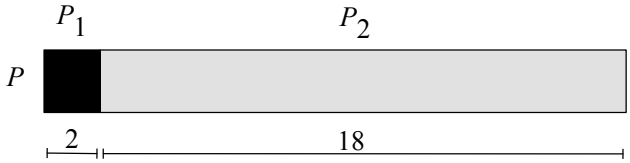


Fig. 2.19: P consists of a non-parallelizable P_1 and a parallelizable P_2. On one processing unit, P_1 runs 2 minutes and P_2 runs 18 minutes.

The clues that the above examples brought to light are recapitulated as follows: In general, a program P executed by a parallel computer can be split into two parts,

- part P_1 which *does not* benefit from multiple processing units, and
- part P_2 which *does benefit* from multiple processing units;
- besides P_2's benefit, also the sequential execution times of P_1 and P_2 influence the parallel execution time of the whole P (and, consequently, P's speedup).

Derivation

We will now assess quantitatively how the speedup of P depends on P_1's and P_2's sequential execution times and their amenability to parallelization and exploitation of multiple processing units.

Let $T_{\text{seq}}(P)$ be the sequential execution time of P. Because $P = P_1P_2$, a sequence of parts P_1 and P_2, we have

$$T_{\text{seq}}(P) = T_{\text{seq}}(P_1) + T_{\text{seq}}(P_2),$$

where $T_{\text{seq}}(P_1)$ and $T_{\text{seq}}(P_2)$ are the sequential execution times of P_1 and P_2, respectively (see Fig. 2.20).

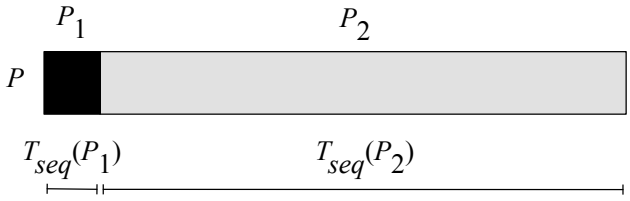


Fig. 2.20: P consists of a non-parallelizable P_1 and a parallelizable P_2. On a single processing unit P_1 requires $T_{\text{seq}}(P_1)$ time and P_2 requires $T_{\text{seq}}(P_2)$ time to complete.

When we actually employ additional processing units in the parallel execution of P, it is the execution of P_2 that is sped up by some factor $s > 1$, while the execution of P_1 does not benefit from additional processing units. In other words, the execution time of P_2 is reduced from $T_{\text{seq}}(P_2)$ to $\frac{1}{s}T_{\text{seq}}(P_2)$, while the execution time of P_1 remains the same, $T_{\text{seq}}(P_1)$. So, after the employment of additional processing units the parallel execution time $T_{\text{seq}}(P)$ of the whole program P is

$$T_{\text{par}}(P) = T_{\text{seq}}(P_1) + \frac{1}{s}T_{\text{seq}}(P_2).$$

The speedup $S(P)$ of the whole program P can now be computed from definition,

$$S(P) = \frac{T_{\text{seq}}(P)}{T_{\text{par}}(P)}.$$

We could stop here; however, it is usual to express $S(P)$ in terms of b, the *fraction* of $T_{\text{seq}}(P)$ during which parallelization of P is beneficial. In our case

$$b = \frac{T_{\text{seq}}(P_2)}{T_{\text{seq}}(P)}.$$

Plugging this in the expression for $S(P)$, we finally obtain the **Amdahl's Law**

$$S(P) = \frac{1}{1-b+\frac{b}{s}}.$$

Some comments on Amdahl's law

Strictly speaking, the speedup in the Amdahl's Law is a function of three variables, P, b and s, so it would be more appropriately denoted by $S(P,b,s)$. Here b is the fraction of the time during which the sequential execution of P *can* benefit from multiple processing units. If multiple processing units are actually available and exploited by P, the part of P that exploits them is sped up by the factor $s > 1$. Since s is only the speedup of a *part* of the program P, the speedup of the *whole* P cannot be larger than s; specifically, it is given by $S(P)$ of the Amdahl's Law.

From the Amdahl's Law we see that

$$S < \frac{1}{1-b},$$

which tells us that a small part of the program which cannot be parallelized will limit the overall speedup available from parallelization. For example, the overall speedup S that the program P in Fig. 2.19 can possibly achieve by parallelizing the part P_2 is bounded above by $S < \frac{1}{1-\frac{18}{20}} = 10$.

Note that in the derivation of the Amdahl's Law nothing is said about the size of the problem instance solved by the program P. It is implicitly assumed that the problem instance remains the same, and that the only thing we carry out is parallelization of P and then application of the parallelized P on the same problem instance. Thus, Amdahl's law only applies to cases where the size of the problem instance is fixed.

Amdahl's law at work

Suppose that 70% of a program execution can be sped up if the program is parallelized and run on 16 processing units instead of one. What is the maximum speedup that can be achieved by the whole program? What is the maximum speedup if we increase the number of processing units to 32, then to 64, and then to 128?

In this case we have $b = 0.7$, the fraction of the sequential execution that can be parallelized; and $1 - b = 0.3$, the fraction of calculation that cannot be parallelized. The speedup of the parallelizable fraction is s. Of course, $s \leqslant p$, where p is the umber of processing units. By Amdahl's Law the speedup of the whole program is

$$S = \frac{1}{1 - b + \frac{b}{s}} = \frac{1}{0.3 + \frac{0.7}{s}} \leqslant \frac{1}{0.3 + \frac{0.7}{16}} = 2.91.$$

If we double the number of processing units to 32 we find that the maximum achievable speedup is 3.11:

$$S = \frac{1}{1 - b + \frac{b}{s}} = \frac{1}{0.3 + \frac{0.7}{s}} \leqslant \frac{1}{0.3 + \frac{0.7}{32}} = 3.11\,,$$

and if we double it once again to 64 processing units, the maximum achievable speedup becomes 3.22:

$$S = \frac{1}{1 - b + \frac{b}{s}} = \frac{1}{0.3 + \frac{0.7}{s}} \leqslant \frac{1}{0.3 + \frac{0.7}{64}} = 3.22.$$

Finally, if we double the number of processing units even to 128, the maximum speedup we can achieve is

$$S = \frac{1}{1 - b + \frac{b}{s}} = \frac{1}{0.3 + \frac{0.7}{s}} \leqslant \frac{1}{0.3 + \frac{0.7}{128}} = 3.27.$$

In this case doubling the processing power only slightly improves the speedup. Therefore, using more processing units is not necessarily the optimal approach.

Note that this complies with actual speedups of realistic programs as we have depicted in Fig. 2.18.

⋆ A generalization of Amdahl's law

Until now we assumed that there are just two parts of of a given program, of which one cannot benefit from multiple processing units and the other can. We now assume that the program is a sequence of three parts, each of which could benefit from multiple processing units. Our goal is to derive the speedup of the whole program when the program is executed by multiple processing units.

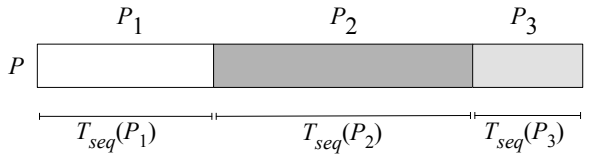


Fig. 2.21: P consists of three differently parallelizable parts.

So let $P = P_1P_2P_3$ be a program which is a sequence of three parts P_1, P_2, and P_3. See Fig. 2.21. Let $T_{\text{seq}}(P_1)$ be the time during which the sequential execution of P spends executing part P_1. Similarly we define $T_{\text{seq}}(P_2)$ and $T_{\text{seq}}(P_3)$. Then the sequential execution time of P is

$$T_{\text{seq}}(P) = T_{\text{seq}}(P_1) + T_{\text{seq}}(P_2) + T_{\text{seq}}(P_3).$$

But we want to run P on a parallel computer. Suppose that the analysis of P shows that P_1 could be parallelized and sped up on the parallel machine by factor $s_1 > 1$. Similarly, P_2 and P_3 could be sped up by factors $s_2 > 1$ and $s_3 > 1$, respectively. So we parallelize P by parallelizing each of the three parts P_1, P_2, and P_3, and run P on the parallel machine. The parallel execution of P takes $T_{\text{par}}(P)$ time, where $T_{\text{par}}(P) = T_{\text{par}}(P_1) + T_{\text{par}}(P_2) + T_{\text{par}}(P_3)$. But $T_{\text{par}}(P_1) = \frac{1}{s_1} T_{\text{seq}}(P_1)$, and similarly for $T_{\text{par}}(P_2)$ and $T_{\text{par}}(P_3)$. It follows that

$$T_{\text{par}}(P) = \frac{1}{s_1} T_{\text{seq}}(P_1) + \frac{1}{s_2} T_{\text{seq}}(P_2) + \frac{1}{s_3} T_{\text{seq}}(P_3).$$

Now the speedup of P can easily be computed from its definition, $S(P) = \frac{T_{\text{seq}}(P)}{T_{\text{par}}(P)}$.

We can obtain a more informative expression for $S(P)$. Let b_1 be the *fraction* of $T_{\text{seq}}(P)$ during which the sequential execution of P executes P_1; that is, $b_1 = \frac{T_{\text{seq}}(P_1)}{T_{\text{seq}}(P)}$. Similarly we define b_2 and b_3. Applying this in the definition of $S(P)$ we obtain

$$S(P) = \frac{T_{\text{seq}}(P)}{T_{\text{par}}(P)} = \frac{1}{\frac{b_1}{s_1} + \frac{b_2}{s_2} + \frac{b_3}{s_3}}.$$

Generalization to programs which are sequences of arbitrary number of parts P_i is straightforward. In reality, programs typically consist of several parallelizable parts and several non-parallelizable (serial) parts. We easily handle this by setting $s_i \geqslant 1$.

2.7 Exercises

1. How many pairwise interactions must be computed when solving the n-body problem if we assume that interactions are symmetric?
2. Give an intuitive explanation why $T_{\text{par}} \leqslant T_{\text{seq}} \leqslant p \cdot T_{\text{par}}$, where T_{par} and T_{seq} are the parallel and sequential execution times of a program, respectively, and p is the number of processing units used during the parallel execution.
3. Can you estimate the number of different network topologies capable of interconnecting p processing units P_i and m memory modules M_j? Assume that each topology should provide, for every pair (P_i,M_j), a path between P_i and M_j.
4. Let P be an algorithm for solving a problem Π on CRCW-PRAM(p). According to Theorem 2.1, the execution of P on EREW-PRAM(p) will be at most $O(\log p)$-times slower than on CRCW-PRAM(p). Now suppose that $p = \text{poly}(n)$, where n is the size of a problem instance. Prove that $\log p = O(\log n)$.
5. Prove that the sum $a_k \log^k n + a_{k-1} \log^{k-1} n + \cdots + a_0$ is asymptotically bounded above by $O(\log^k n)$.
6. Prove that $O(n/\log n)$ processing units suffice to sum up n numbers in $O(\log n)$ parallel time.
   Hint: Assume that the numbers are summed up with a tree-like parallel algorithm described in Example 2.1. Use Brent's Theorem with $W = n - 1$ and $T = \log n$ and observe that by reducing the number of processing units to $p := n \log n$, the tree-like parallel algorithm will retain its $O(\log n)$ parallel time complexity.
7. True or false:
   a) The definition of the parallel execution time is: "execution time = computation time + communication time + idle time."
   b) A simple model of the communication time is:"communication time = set-up time + data transfer time."
   c) Suppose that the execution time of a program on a single processor is T_1, and the execution time of the same parallelized program on p processors is T_p. Then, the speedup and efficiency are $S = T_1/T_p$ and $E = S/p$, respectively.
   d) If speedup $S < p$ then $E > 1$.
8. True or false:
   a) If processing units are identical, then in order to minimize parallel execution time, the work (or, computational load) of a parallel program should be partitioned into equal parts and distributed among the processing units.
   b) If processing units differ in their computational power, then in order to minimize parallel execution time, the work (or, computational load) of a parallel program should be distributed evenly among the processing units.
   c) Searching for such distributions is called **load balancing**.
9. Why must be the load of a parallel program evenly distributed among processors?
10. Determine the bisection bandwidths of 1D-mesh (chain of computers with bidirectional connections), 2D-mesh, 3D-mesh, and the hypercube.

11. Let a program P be composed of a part R that can be ideally parallelized, and of a sequential part S; that is, $P = RS$. On a single processor, S takes 10% of the total execution time and during the remaining 90% of time R could run in parallel.
    a) What is the maximal speedup reachable with unlimited number of processors?
    b) How is this law called?
12. **Moore's law** states that computer performance doubles every 1.5 year. Suppose that the current computer performance is Perf $= 10^{13}$. When will be, according to this law, 10 times greater (that is, $10 \times$ Perf)?
13. A problem Π comprises two subproblems, Π_1 and Π_2, which are solved by programs P_1 and P_2, respectively. The program P_1 would run 1000 seconds on the computer C_1 and 2000 seconds on the computer C_2, while P_2 would require 2000 and 3000 seconds on C_1 and C_2, respectively. The computers are connected by a 1000-km long optical fiber link capable of transferring data at 100 MB/sec with 10 msec latency. The programs can execute concurrently but must transfer either a) 10 MB of data 20,000 times or b) 1 MB of data twice during the execution. What is the best configuration and approximate runtimes in cases a) and b)?

2.8 Bibliographical notes

In presenting the topics in this Chapter we have strongly leaned on Trobec at al. [26] and Atallah and Blanton [3]. On the computational models of sequental computation see Robič [22]. Interconnection networks are discussed in great detail in Dally and Towles [6], Duato et al. [7], Trobec [25] and Trobec at al. [26]. The dependence of execution times of real world parallel applications on the performance of the interconnection networks is discussed in Grama et al. [12].

Part II
Programming

In Part II is devoted to programming of parallel computers. It is designed in a way that every reader can exploit the parallelism of its own computer, either on multi-cores with shared memory, or on a set of interconnected computers, or on a graphic processing units. The knowledge and experience obtained can be beneficial in eventual further, more advanced applications, which will run on many-core computers, computing clusters, or heterogeneous computers with computing accelerators.

We start in Chapter 3 with multi-core and shared memory multiprocessors, which is the architecture of almost all contemporary computers, that are likely the easiest to program with an adequate methodology. The programming such systems is introduced using OpenMP, a widely used and ever expanding application programming interface well suited for the implementation of multithreaded programs. It is shown how the combination of properly designed compiler directives and library functions can provide a programming environment where the programmer can focus mostly on the program and algorithms and less on the details of the underlying computer architecture. A lot of practical examples are provided, which help the reader to understand the basic principles and to get a further motivation for fully exploiting available computing resources.

Next, in Chapter 4, distributed memory computers are considered. They cannot communicate through the shared memory therefore messages are used for the coordination of parallel tasks that run on geographically distributed but interconnected processors. Definition of processes with their management and communication are well defined by a platform-independent message passing interface (MPI) specification. The MPI library is introduced from the practical point of view, with basic set of operations that enable the implementation of parallel programs. Simple example programs should serve as an aid for a smooth start of using MPI and as motivation for developing more complex applications.

Finally, in Chapter 5, we provide an introduction to the concepts of massively parallel programming on GPUs and heterogeneous systems. Almost all contemporary desktop computers are multi-core processor with a GPU units. Thus we need a programming environment in which a programmer can write programs and run them on either a GPU, or on a multi-core CPU, or on both. Again, several practical examples are given that help and assist the readers in acquiring knowledge and experience in programming GPUs, using OPenCL environment.

Chapter 3
Programming multi-core and shared memory multiprocessors using OpenMP

Abstract Of many different parallel and distributed systems, multi-core and shared memory multiprocessors are most likely the easiest to program if only the right approach is taken. In this chapter programming such systems is introduced using OpenMP, a widely used and ever expanding application programming interface well suited for the implementation of multithreaded programs. It is shown how the combination of properly designed compiler directives and library functions can provide a programming environment where the programmer can focus mostly on the program and algorithms and less on the details of the underlying computer system.

3.1 Shared memory programming model

From the programmer's point of view, a model of a **shared memory multiprocessor** contains a number of independent processors all sharing a single main memory as shown in Figure 3.1. Each processor can directly access any data location in the main memory and at any time different processors can execute different instructions on different data since each processor is driven by its own control unit. Using *Flynn's taxonomy* of parallel systems, this model is referred to as *MIMD*, i.e., multiple instruction multiple data.

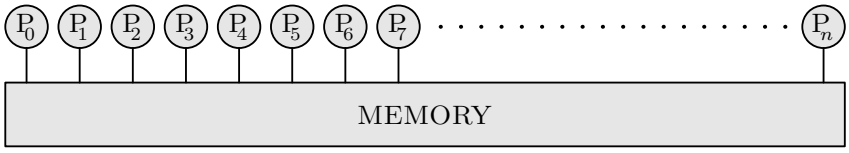


Fig. 3.1: A model of a shared-memory multiprocessor.

Although of paramount importance when the parallel system is to be fully utilized to achieve the maximum speedup possible, many details like cache or the in-

ternal structure of the main memory are left out of this model to keep it simple and general. Using a simple and general model also simplifies the design and implementation of portable programs that can be optimized for a particular system once the parallel system specifications are known.

Most modern CPUs are **multi-core processors** and therefore consist of a number of independent processing units called **cores**. Moreover, these CPUs support (simultaneous) **multithreading** (SMT) so that each core can (almost) simultaneously execute multiple independent streams of instructions called **threads**. To a programmer each core within each processor acts as several **logical cores** each able to run its own program or a thread within a program independently.

Today, mobile, desktop and server CPUs typically contain 2 to 24 cores and with multithreading support, they can run 4 to 48 threads simultaneously. For instance, a dual core mobile Intel i7 processor with hyper-threading (Intel's SMT) consists of 2 (physical) cores and thus provides 4 logical cores. Likewise, a quad-core Intel Xeon processor with hyper-threading provides 8 logical cores and a system with two such CPUs provides 16 logical cores as shown in Figure 3.2. If the common use of certain resources like bus or cache is set aside, each logical core can execute its own thread independently. Regardless of the physical implementation, a programmer can assume that such system contains 16 logical cores each acting as an individual processor as shown in Figure 3.1 where $n = 16$.

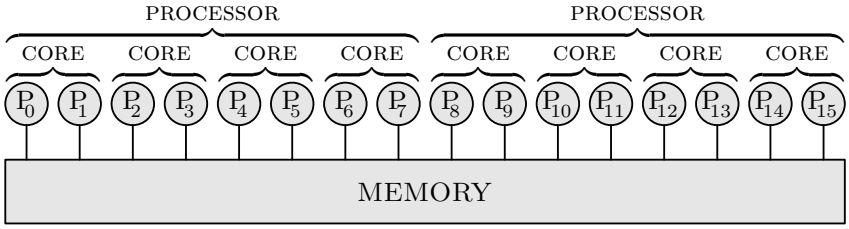


Fig. 3.2: A parallel system with two quad-core CPUs supporting simultaneous multithreading contains 16 logical cores all connected to the same memory.

Apart from multi-core CPUs, *manycore processors* comprising tens or hundreds of physical cores are also available. Intel Xeon Phi, for instance, provides 60 to 72 physical cores able to run 240 to 288 threads simultaneously.

The ability of modern systems to execute multiple threads simultaneously using different processors or (logical) cores comes with a price. As individual threads can access any memory location in the main memory and to execute instruction streams independently may result in a **race condition**, i.e., a situation where the result depends on precise timing of read and write accesses to the same location in the main memory. Assume, for instance, that two threads must increase the value stored at the same memory location, one by 1 and the other by 2 so that in the end

it is increased by 3. Each thread reads the value, increases it and writes it back. If these three instructions are first performed by one thread and then by the other, the correct result is produced. But because threads execute instructions independently, the sequences of these three instructions executed by each thread may overlap in time as illustrated in Figure 3.3. In such situations, the result is both incorrect and undefined: in either case, the value in the memory will be increased by either 1 or 2 but not by 1 and 2.

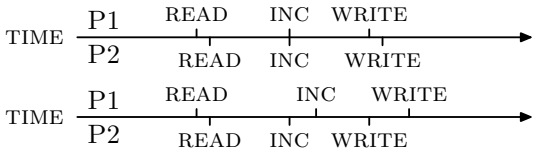


Fig. 3.3: Two examples of a race condition when two threads attempt to increase the the value at the same location in the main memory.

To avoid the race condition, exclusive access to the shared address in the main memory must be ensured using some mechanism like **locking** using semaphores or **atomic access** using read-modify-write instructions. If locking is used, each thread must lock the access to the shared memory location before modifying it and unlock it afterwards as illustrated in Figure 3.4. If a thread attempts to lock something that the other thread has already locked, it must wait until the other thread unlocks it. This approach forces one thread to wait but guarantees the correct result.

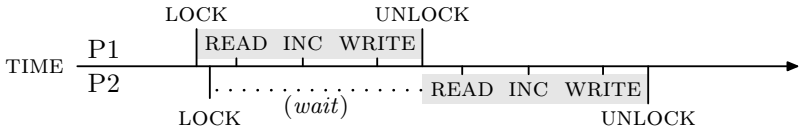


Fig. 3.4: Preventing race conditions as illustrated in Figure 3.3 using locking.

The peripheral devices are not shown in Figures 3.1 and 3.2. It is usually assumed that all threads can access all peripheral devices but it is again up to software to resolve which thread can access each device at any given time.

3.2 Using OpenMP to write multithreaded programs

A parallel program running on a shared memory multiprocessor usually consists of multiple threads. The number of threads may vary during program execution but at any time each thread is being executed on one logical core. If there are less

threads than logical cores, some logical cores are kept idle and the system is not fully utilized. If there are more threads than logical cores, the operating system applies multitasking among threads running on the same logical cores. During program execution the operating system may perform *load balancing*, i.e., it may migrate threads from one logical core to another in an attempt to keep all logical cores equally utilized.

A multithreaded program can be written in different programming languages using many different libraries and frameworks. On UNIX, for instance, one can use `pthreads` in almost any decent programming language, but the resulting program is littered with low-level details that the compiler could have taken care of, and is not portable. Hence, it is better to use something dedicated to writing parallel programs.

One such thing is OpenMP, a parallel programming environment best suitable for writing parallel programs that are to be run on shared memory systems. It is not yet another programming language but an add-on to an existing language, usually Fortran or C/C++. In this book, OpenMP atop of C will be used.

The application programming interface (API) of OpenMP is a collection of

- compiler directives,
- supporting functions, and
- shell variables.

OpenMP compiler directives tell the compiler about the parallelism in the source code and provide instructions for generating the parallel code, i.e., the multithreaded translation of the source code. In C/C++, directives are always expressed as `#pragmas`. Supporting functions enable programmers to exploit and control the parallelism during the execution of a program. Shell variables permit tunning of compiled programs to a particular parallel system.

3.2.1 Compiling and running an OpenMP program

To illustrate different kinds of OpenMP API elements we will start with a simple program in Listing 3.1.

```
#include <stdio.h>
#include <omp.h>

int main() {
  printf ("Hello, world:");
  #pragma omp parallel
    printf (" %d", omp_get_thread_num ());
  printf ("\n");
  return 0;
}
```

Listing 3.1: “Hello world” program, OpenMP style.

This program starts as a single thread that first prints out the salutation. Once the execution reaches the `omp parallel` directive, several additional threads are

OpenMP: parallel regions

A parallel region within a program is specified as

`#pragma omp parallel` *[clause [[,] clause] . . .]*
structured-block

A team of threads is formed and the thread that encountered the `omp parallel` directive becomes the **master thread** within this team. The *structured-block* is executed by every thread in the team. It is either a single statement, possibly compound, with a single entry at the top and a single exit at the bottom, or another OpenMP construct. At the end there is an implicit **barrier**, i.e., only after all threads have finished, the threads created by this directive are terminated and only the master resumes execution.

A parallel region might be refined by a list of *clause*s, for instance

- `num_threads(`*integer*`)` specifies the number of threads that should execute *structured-block* in parallel.

Some other *clause*s applicable to `omp parallel` will be introduced later.

created alongside the existing one. All threads, the initial thread and the newly created threads, together form a *team of threads*. Each thread in the newly established team of threads executes the statement immediately following the directive: in this example it just prints out its unique thread number obtained by calling OpenMP function `omp_get_thread_num`. When all threads have done that, threads created by the `omp parallel` directive are terminated and the program continues as a single thread that prints out a single new line character and terminates the program by executing `return 0`.

To compile and run the program shown in Listing 3.1 using GNU GCC C/C++ compiler, use the command-line option `-fopenmp` as follows:

```
$ gcc -fopenmp -o hello-world hello-world.c
$ env OMP_NUM_THREADS=8 ./hello-world
```

(See Appendix A for instructions on how to make OpenMP operational on Linux, macOS and MS Windows.)

In this program the number of threads is not specified explicitly. Hence, the number of threads matches the value of the shell variable `OMP_NUM_THREADS`. Setting the value of `OMP_NUM_THREADS` to 8, the program might print out

```
Hello, world: 2 5 1 7 6 0 3 4
```

Without `OMP_NUM_THREADS` being set, the program would set the number of threads to match the number of logical cores threads can run on. For instance, on a CPU with 2 cores and hyper-threading, 4 threads would be used and a permutation of numbers from 0 to 3 would be printed out.

Once the threads are started, it is up to a particular OpenMP implementation and especially the underlying operating system to carry out scheduling and to resolve

OpenMP: controlling the number of threads

Once a program is compiled, the number of threads can be controlled using the following shell variables:

- `OMP_NUM_THREADS` *comma-separated-list-of-positive-integers*
- `OMP_THREAD_LIMIT` *positive-integer*

The first one sets the number of threads the program should use (or how many threads should be used at every nested level of parallel execution). The second one limits the number of threads a program can use (and takes the precedence over `OMP_NUM_THREADS`).

Within a program, the following functions can be used to control the number of threads:

- `void omp_set_num_threads()` sets the number of threads used in the subsequent parallel regions without explicit specification of the number of threads;
- `int omp_get_num_threads()` returns the number of threads in the current team relating to the innermost enclosing parallel region;
- `int omp_get_max_threads()` returns the maximal number of threads available to the subsequent parallel regions;
- `int omp_get_thread_num()` returns the thread number of the calling thread within the current team of threads.

competition for the single standard output the permutation is printed on. Hence, if the program is run several times, a different permutation of thread numbers will most likely be printed out each time. Try it.

3.2.2 Monitoring an OpenMP program

During the design, development and debugging of parallel programs reasoning about parallel algorithms and how to encode them better rarely suffices. To understand how an OpenMP program actually runs on a multi-core system, it is best to monitor and measure the performance of the program. Even more, this is the simplest and the most reliable way to know how many cores your program actually runs on.

Let us use the program in Listing 3.2 as an illustration. The program starts several threads, each of them printing out one Fibonacci number computed using the naive and time-consuming recursive algorithm.

On most operating systems it is usually easy to measure the running time of a program execution. For instance, compiling the above program and running it using `time` utility on Linux as

```
$ gcc -fopenmp -O2 -o fibonacci fibonacci.c
```

```
#include <stdio.h>
#include <omp.h>

long fib (int n) { return (n < 2 ? 1 : fib (n - 1) + fib (n - 2)); }

int main () {
  int n = 45;
  #pragma omp parallel
    {
      int t = omp_get_thread_num ();
      printf ("%d: %ld\n", t, fib (n + t));
    }
  return 0;
}
```

Listing 3.2: Computing some Fibonacci numbers.

```
$ env OMP_NUM_THREADS=8 time ./fibonacci
```

yields some Fibonacci numbers and then, as the last line of output, the information about the program's running time:

```
        106.46 real       298.45 user         0.29 sys
```

(See Appendix A for instructions on how to measure time and monitor the execution of a program on Linux, macOS and MS Windows.)

The user and system time amount to the total time that all logical cores together spent executing the program. In the example above, the sum of the user and system time is bigger than the real time, i.e., the elapsed or wall-clock time. Hence, various parts of the program must have run on several logical cores simultaneously.

Most operating systems provide system monitors that among other metrics show the amount of computation performed by individual cores. This might be very informative during OpenMP program development, but be careful as most system monitor reports the overall load on an individual logical core, i.e., load of all programs running on a logical core.

Using a system monitor while the program shown in Listing 3.2 is run on otherwise idle system, one can observe the load on individual logical cores during program execution. As threads finish one after another, one can observe how the load on individual logical cores drops as the execution proceeds. Towards the end of execution, with only one thread remaining, it can be seen how the operating system occasionally migrates the last thread from one logical core to another.

3.3 Parallelization of loops

Most CPU-intensive programs for solving scientific or technical problems spend most of their time running loops so it is best to start with some examples illustrating what OpenMP provides for the efficient and portable implementation of **parallel loops**.

3.3.1 Parallelizing loops with independent iterations

To avoid obscuring the explanation of parallel loops in OpenMP with unnecessary details, we start with a trivial example of a loop parallelization: consider printing out all integers from 1 to some user specified value, say *max*, in no particular order. The parallel program is shown in Listing 3.3.

```
1  #include <stdio.h>
2  #include <omp.h>
3
4  int main (int argc, char *argv[]) {
5    int max; sscanf (argv[1], "%d", &max);
6    #pragma omp parallel for
7      for (int i = 1; i <= max; i++)
8        printf ("%d: %d\n", omp_get_thread_num (), i);
9    return 0;
10 }
```

Listing 3.3: Printing out all integers from 1 to *max* in no particular order.

The program in Listing 3.3 starts as a single initial thread. The value *max* is read and stored in variable `max`. The execution then reaches the most important part of the program, namely the `for` loop which actually prints out the numbers (each preceded by the number of a thread that prints it out). But the `omp parallel for` directive in line 6 specifies that the `for` loop must be executed in parallel, i.e., its iterations must be divided among and executed by multiple threads running on all available processing units. Hence, a number of **slave threads** is created, one per each available processing unit or as specified explicitly (minus one that the initial thread runs on). The initial thread becomes the **master thread** and together with the newly created slave threads the *team of threads* is formed. Then,

- iterations of the parallel `for` loop are divided among threads where each iteration is executed by the thread it has been assigned to, and
- once all iterations have been executed, all threads in the team are synchronized at the implicit barrier at the end of the parallel `for` loop and all slave threads are terminated.

Finally, the execution proceeds sequentially and the master thread terminates the program by executing `return 0`. The execution of the program in Listing 3.3 is illustrated in Figure 3.5.

Several observation must be made regarding the program in Listing 3.3 (and execution of parallel `for` loops in general). First, the program in Listing 3.3 does not specify how the iterations should be divided among threads (as explicit scheduling of iterations will be described later). In such cases, most OpenMP implementations divide the entire iteration space into chunks where each chunk containing a subinterval of all iterations is executed by one thread. Note, however, that this must not be the case as if left unspecified, it is up to a particular OpenMP implementation to do as it likes.

OpenMP: parallel loops

A parallel `for` loops are declared as

`#pragma omp for` *[clause [[,] clause] . . .]*
for-loops

This directive, which must be used within a parallel region, specifies that iterations of one or more nested `for` loops will be executed by the team of threads within the parallel region (`omp parallel for` is a shorthand for writing a `for` loop that itself encompasses the entire parallel region). Each `for` loop among *for-loops* associated with the `omp for` directive must be in the **canonical form**. In C, that means that

- the loop variable is made private to each thread in the team and must be either (unsigned) integer or a pointer,
- the loop variable should not be modified during the execution of any iteration;
- the condition in the `for` loop must be a simple relational expression,
- the increment in the `for` loop must specify a change by constant additive expression;
- the number of iterations of all associated loops must be known before the start of the outermost `for` loop.

A *clause* is a specification that further describes a parallel loop, for instance

- `collapse(`*integer*`)` specifies how many outermost `for` loops of *for-loops* are associated with the directive and thus parallelized together;
- `nowait` eliminates the implicit barrier and thus synchronization at the end of *for-loops*.

Some other clauses applicable to `omp parallel for` will be introduced later.

Second, once the iteration space is divided into chunks, all iterations of an individual chunk are executed sequentially, one iteration after another. And third, the

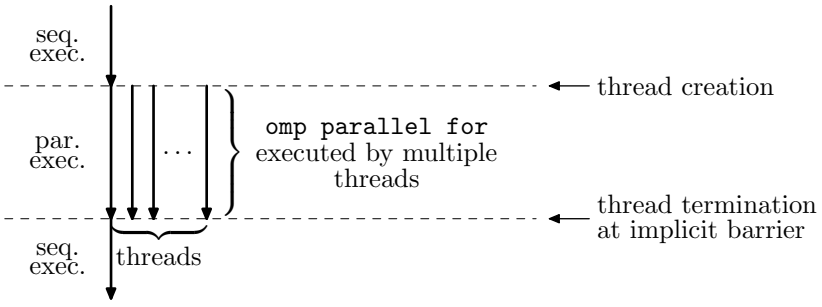


Fig. 3.5: Execution of the program for printing out integers as implemented in Listing 3.3.

OpenMP: data sharing

Various data sharing clauses might be used in `omp parallel` directive to specify whether and how data are shared among threads:

- `shared(`*list*`)` specifies that each variable in the *list* is shared by all threads in a team, i.e., all threads share the same copy of the variable;
- `private(`*list*`)` specifies that each variable in the *list* is private to each thread in a team, i.e., each thread has its own local copy of the variable;
- `firstprivate(`*list*`)` is like `private` but each variable listed is initialized with the value it contained when the parallel region was encountered;
- `lastprivate(`*list*`)` is like `private` but when the parallel region ends each variable listed is updated with its final value within the parallel region.

No variable listed in these clauses can be a part of another variable.

If not specified otherwise,

- automatic variables declared outside a parallel construct are shared,
- automatic variables declared within a parallel construct are private,
- static and dynamically allocated variables are shared.

Race conditions, e.g., resulting from different life times of `lastprivate` variables or updating shared variables, must be avoided explicitly by using OpenMP constructs described later on.

parallel `for` loop variable `i` is made private in each thread executing a chunk of iterations as each thread must have its own copy of `i`. On the other hand, variable `max` can be shared by all threads as it is set before and is only read within the parallel region.

However, the most important detail that must be paid attention to is that the overall task of printing out all integers from 1 to *max* in no particular order can be divided into *N totally independent* subtasks of *almost the same size*. In such cases, the parallelization is trivial.

As the access to the standard output is serialized, printing out integers does not happen as parallel as it might seen. Therefore an example of truly parallel computation follows.

Example 3.1. Vector addition

Consider vector addition. The function implementing it is shown in Listing 3.4. Write a program for testing it. As vector addition is not a complex computation at all, use long vectors and perform a large number of vector additions to measure and monitor it.

The structure of function `vectAdd` is very similar to the program for printing out integers shown in Listing 3.3: a simple parallel `for` loop where the result of one iteration is completely independent of the results produced by other loops. Even more, different iterations access different array elements, i.e., they read from and write to completely different memory locations. Hence, no race conditions can occur. □

```
double* vectAdd (double *c, double *a, double *b, int n) {
  #pragma omp parallel for
    for (int i = 0; i < n; i++)
      c[i] = a[i] + b[i];
  return c;
}
```

Listing 3.4: Parallel vector addition.

Consider now printing out all pairs of integers from 1 to *max* in no particular order, something that calls for two nested `for` loops. As all iterations of both nested loops are independent, either loop can be parallelized while the other is not. This is achieved by placing the `omp parallel for` directive in front of the loop targeted for parallelization. For instance, the program with the outer `for` loop parallelized is shown in Listing 3.5.

```
#include <stdio.h>
#include <omp.h>

int main (int argc, char *argv[]) {
  int max; sscanf (argv[1], "%d", &max);
  #pragma omp parallel for
    for (int i = 1; i <= max; i++)
      for (int j = 1; j <= max; j++)
        printf ("%d: (%d,%d)\n", omp_get_thread_num (), i, j);
  return 0;
}
```

Listing 3.5: Printing out all pairs of integers from 1 to *max* in no particular order by parallelizing the outermost `for` loop only.

Assume all pairs of integers from 1 to *max* are arranged in a square table. If 4 threads are used and $max = 6$, each iteration of the parallelized outer `for` loop prints out a few lines of the table as illustrated in Figure 3.6 (a). Note that the first two threads are assigned twice as much work than the other two threads which, if run on 4 logical cores, will have to wait idle until the first two complete as well.

However, there are two other ways of parallelizing nested loops. First, the two nested `for` loops can be collapsed in order to be parallelized together using clause `collapse(2)` as shown in Listing 3.6.

```
#include <stdio.h>
#include <omp.h>

int main (int argc, char *argv[]) {
  int max; sscanf (argv[1], "%d", &max);
  #pragma omp parallel for collapse(2)
    for (int i = 1; i <= max; i++)
      for (int j = 1; j <= max; j++)
        printf ("%d: (%d,%d)\n", omp_get_thread_num (), i, j);
  return 0;
}
```

Listing 3.6: Printing out all pairs of integers from 1 to *max* in no particular order by parallelizing both `for` loops together.

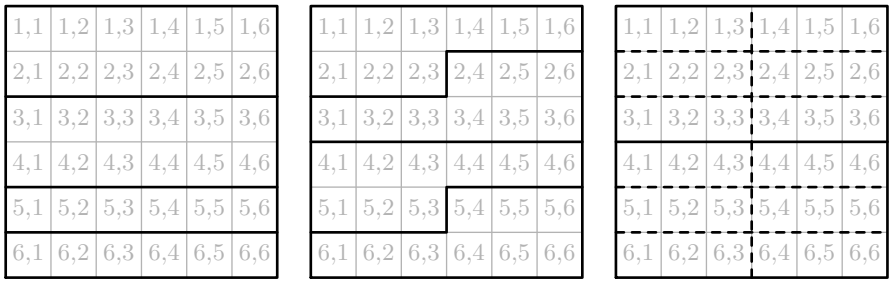


Fig. 3.6: Partition of the problem domain when all pairs of integers from 1 to 6 must be printed using 4 threads: (a) if only the outer `for` loop is parallelized, (b) if both `for` loops are parallelized together, and (c) if both `for` loops are parallelized separately.

Because of the clause `collapse(2)` in line 6 the compiler merges the two nested `for` loops into one and parallelizes the resulting single loop. The outer `for` loop running from 1 to *max* and *max* inner `for` loops running from 1 to *max* as well, are replaced by a single loop running from 1 to max^2. All max^2 iterations are divided among available threads together. As only one loop is parallelized, i.e., the one that comprises iterations of both nested `for` loops, the execution of the program in Listing 3.6 still follows the pattern illustrated in Figure 3.5. For instance, if $max = 6$, all 36 iterations of the collapsed single loop are divided among 4 thread as shown in Figure 3.6 (b). Compared with the program in Listing 3.5, the work is more evenly distributed among threads.

The other method of parallelizing nested loops is by parallelizing each `for` loop separately as shown in Listing 3.7.

```
1  #include <stdio.h>
2  #include <omp.h>
3
4  int main (int argc, char *argv[]) {
5    int max; sscanf (argv[1], "%d", &max);
6    #pragma omp parallel for
7      for (int i = 1; i <= max; i++) {
8        #pragma omp parallel for
9          for (int j = 1; j <= max; j++) {
10           printf ("%d: (%d,%d)\n", omp_get_thread_num (), i, j);
11         }
12     }
13   return 0;
14 }
```

Listing 3.7: Printing out all pairs of integers from 1 to max in no particular order by parallelizing each nested `for` loop separately.

To have one parallel region within the other as shown in Listing 3.7 active at the same time, nesting of parallel regions must be enabled first. This is achieved by

OpenMP: nested parallelism

Nested parallelism is enabled or disabled by setting the shell variable

- `OMP_NESTED` *nested*

where *nested* is either `true` or `false`. Within a program, this can be achieved usign the following two functions:

- `void omp_set_nested(int nested)` enables or disables nested parallelism;
- `int omp_get_nested()` tells whether nested parallelism is enabled or disabled.

The number of threads at each nested level can be set by calling function `omp_set_num_threads` or by setting `OMP_NUM_THREADS`. In the latter case, if a list of integers is given each integer specifies the number of threads at a successive nesting level.

calling `omp_set_nested(1)` before `mtxMul` is called or by setting `OMP_NESTED` to `true`. Once nesting is activated, iterations of both loops are executed in parallel separately as illustrated in Figure 3.7. Compare Figures 3.5 and 3.7 and note how many more threads are created and terminated in the latter, i.e., if nested loops are parallelized separately.

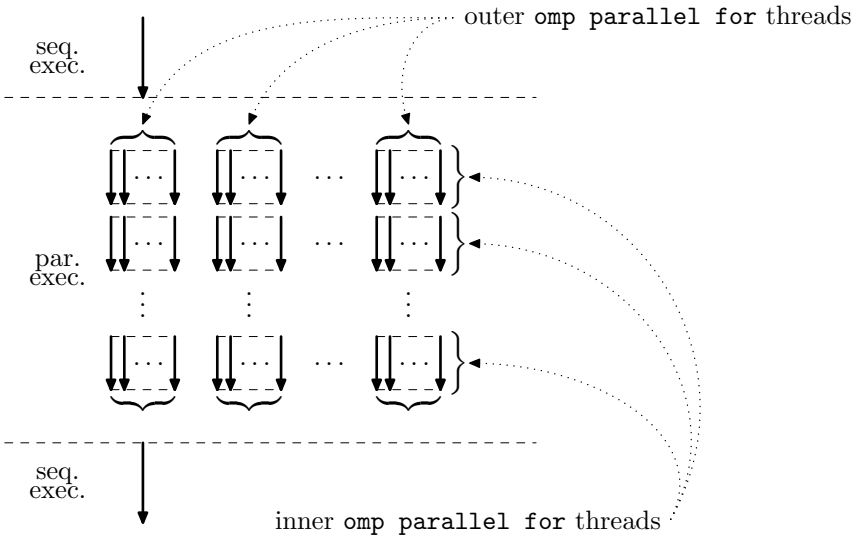


Fig. 3.7: The execution of the program for printing out all pairs of integers using separately parallelized nested loops as implemented in Listing 3.7.

By setting `OMP_NUM_THREADS=2,2` and running the program in Listing 3.7, the team of two (outer) threads is established to execute 3 iterations of the outer loop each as they would even if nesting was disabled. Each iteration of the outer loop must compute one line of the table and thus establishes a team of two (inner) threads to execute 3 iterations of the inner loop each. The table of pairs to be printed out is divided among threads as shown in Figure 1.6 (c). However, be careful while interpreting the output: threads of every iteration of the inner loop are counted from 0 onward because function `omp_get_thread_num` always returns the thread number relative to its team.

Example 3.2. Matrix multiplication

Another example from linear algebra is matrix by matrix multiplication. The classical algorithm, based on the definition, encompasses two nested `for` loops used to compute n^2 independent dot products (and each dot product is computed using yet another, innermost, `for` loop).

Hence, the structure of the matrix multiplication code resembles the code shown in Listings 3.5, 3.6 and 3.7 except that the simple code for printing out the pair of integers is replaced by yet another `for` loop for computing the dot product.

The function implementing multiplication of two square matrices where the two outermost `for` loops are collapsed and parallelized together, is shown in Listing 3.8. As before, write a program for testing it.

```
double **mtxMul (double **c, double **a, double **b, int n) {
  #pragma omp parallel for collapse(2)
    for (int i = 0; i < n; i++)
      for (int j = 0; j < n; j++) {
        c[i][j] = 0.0;
        for (int k = 0; k < n; k++)
          c[i][j] = c[i][j] + a[i][k] * b[k][j];
      }
  return c;
}
```

Listing 3.8: Matrix multiplication where the two outermost loops are parallelized together.

The other way of parallelizing matrix multiplication is shown in Listing 3.9.

```
double **mtxMul (double **c, double **a, double **b, int n) {
  #pragma omp parallel for
    for (int i = 0; i < n; i++)
      #pragma omp parallel for
        for (int j = 0; j < n; j++) {
          c[i][j] = 0.0;
          for (int k = 0; k < n; k++)
            c[i][j] = c[i][j] + a[i][k] * b[k][j];
        }
  return c;
}
```

Listing 3.9: Matrix multiplication where the two outermost loops are parallelized separately.

Writing functions for matrix multiplication where only one of the two outermost `for`loops is parallelized, either outer of inner, is left as an exercise. □.

Example 3.3. Convay's Game of Life

Both examples so far have been taken from linear algebra. Let us now consider something different: Conway's Game of life. It is a zero-player game played on a (finite) plane of square cells, each of which is either "dead" or "alive". At each step in time a new generation of cells arises where

- each live cell with fewer than two neighbours dies of underpopulation,
- each live cell with two or three neighbours lives on,
- each live cell with more than three neighbours dies of overpopulation, and
- each dead cell with three neighbours becomes a live cell.

It is assumed that each cell has eight neighbours, four along its sides and four on its corners.

Once the initial generation is set, all the subsequent generations can be computed. Sometimes the population of live cells die out, sometimes it turns into a static colony, other times it oscillates forever. Even more sophisticated patterns can appear including traveling colonies and colony generators. Figure 3.8 shows an evolution of an oscillating colony on the 10×10 plane.

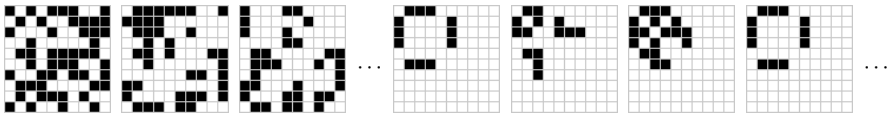


Fig. 3.8: Conway's Game of Life: a particular initial population turns into an oscillating one.

The program for computing Convay's Game of life is too long to be included entirely, but its core is shown in Listing 3.10. To understand it, observe the following:

- variable `gens` contains the number of generations to be computed;
- the current generation is stored in a two dimensional array `plane` containing `size` $\times$ `size` cells;
- the next generation is computed into a two dimensional array `aux_plane` containing `size` $\times$ `size` cells;
- both two dimensional arrays, i.e., `plane` and `aux_plane`, are allocated as a one dimensional array of pointers to rows of two dimensional plane;
- function `neighbors` returns the number of neighbors of the cell in the plane specified by the first argument of size specified by the second argument at position specified by the third and fourth arguments.

Except for the `omp parallel for` directive, the code in Listing 3.10 is the same as if it was written for the sequential execution: the (outermost) `while` loop runs

over all generations to be computed while the inner two loops are used to compute the next generation and store it in `aux_plane` given the current generation in `plane`. More precisely, the rules of the game are implemented in the `switch` statement in lines 6–11: the case for `plane[i][j]==0` implements the rule for dead cells and the case for `plane[i][j]==1` implements the rules for live cells. Once the new generation has been computed, the arrays are swapped so that the generation just computed becomes the current one.

The `omp parallel for` directive in line 2 is used to specify that the iterations of the two `for` loops in lines 3–12 can be performed simultaneously. By inspecting the code it becomes clear that just like in matrix multiplication every iteration of the outer `for` loop computes one row of the plane representing the next generation and that every iteration of the inner loop computes a single cell of the the next generation. As array `plane` is read only and the (i, j)-th iteration of the collapsed loop is the only one writing to the (i, j)-th cell of array `aux_plane`, there can be no race conditions and there are no dependencies among iterations.

The implicit synchronization at the end of the parallelized loop nest is crucial. Without synchronization, if the master thread performed the swap in line 13 before other threads finished the computation within both `for` loops, it would cause all other threads to mess up the computation profoundly.

Finally, instead of parallelizing the two `for` loops together it is also possible to parallelize them separately just like in matrix multiplication. But the outermost loop, i.e., `while` loop, cannot be parallelized as every iteration (except the first one) depends on the result of the previous one. □

3.3.2 Combining the results of parallel iterations

In most cases, however, individual loop iterations aren't entirely independent as they are used to solve a single problem together and thus each iteration contributes its part to the combined solution. Most often then not partial results of different iterations must be combined together.

```
1  while (gens-- > 0) {
2    #pragma omp parallel for collapse(2)
3      for (int i = 0; i < size; i++)
4        for (int j = 0; j < size; j++) {
5          int neighs = neighbors (plane, size, i, j);
6          switch (plane[i][j]) {
7            case 0: aux_plane[i][j] = (neighs == 3);
8                    break;
9            case 1: aux_plane[i][j] = (neighs == 2) || (neighs == 3);
10                   break;
11         }
12       }
13   char **tmp_plane = aux_plane; aux_plane = plane; plane = tmp_plane;
14 }
```

Listing 3.10: Computing generations of Convay's Game of Life.

If integers from the given interval are to be added instead of printed out, all subtasks must somehow cooperate to produce the correct sum. The first parallel solution that comes to mind is shown in Listing 3.11. It uses a single variable `sum` where the result is to be accumulated.

```
1  #include <stdio.h>
2
3  int main (int argc, char *argv[]) {
4    int max; sscanf (argv[1], "%d", &max);
5    int sum = 0;
6    #pragma omp parallel for
7      for (int i = 1; i <= max; i++)
8        sum = sum + i;
9    printf ("%d\n", sum);
10   return 0;
11 }
```

Listing 3.11: Summation of integers from a given interval using a single shared variable — wrong.

Again, iterations of the parallel `for` loop are divided among multiple threads. In all iterations, threads use the same shared variable `sum` on both sides of assignment in line 8, i.e., they read from and write to the same memory location. As illustrated in Figure 3.9 where every box containing `=+` denotes the assignment `sum = sum + i`, the accesses to variable `sum` overlap and the program is very likely to encounter race conditions illustrated in Figure 3.3.

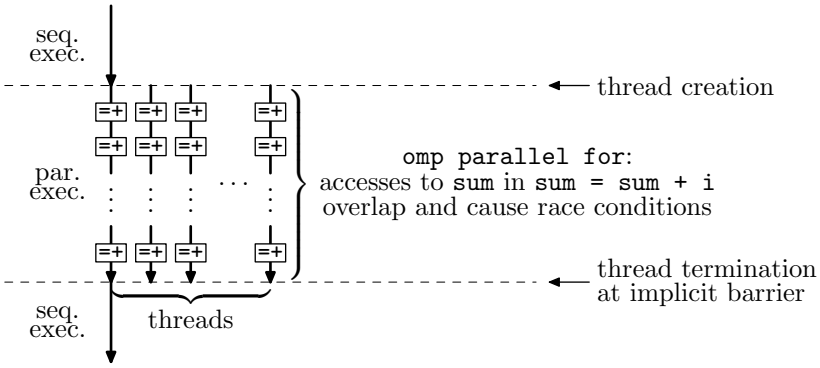


Fig. 3.9: Execution of the summation of integers as implemented in Listing 3.11.

Indeed, if this program is run multiple times using several threads, it is very likely that it will not always produce the same result. In other words, from time to time it will produce the wrong result. Try it.

To avoid race conditions, the assignment `sum = sum + i` can be put inside a **critical section** — a part of a program that is performed by at most one thread at

a time. This is achieved by the `omp critical` directive which is applied to the statement or a block immediately following it. The program using critical sections is shown in Listing 3.12.

```
#include <stdio.h>

int main (int argc, char *argv[]) {
  int max; sscanf (argv[1], "%d", &max);
  int sum = 0;
  #pragma omp parallel for
    for (int i = 1; i <= max; i++)
      #pragma omp critical
        sum = sum + i;
  printf ("%d\n", sum);
  return 0;
}
```

Listing 3.12: Summation of integers from a given interval using a critical section — slow.

The program works correctly because the `omp critical` directive performs locking around the code it contains, i.e., the code that accesses variable `sum`, as illustrated in Figure 3.4 and thus prevents race conditions. However, the use of critical sections in this program makes the program slow because at every moment at most one thread performs the addition and assignment while all other threads are kept waiting as illustrated in Figure 3.10.

It is worth comparing the running times of the programs shown in Listings 3.11 and 3.12. On a fast multi-core processor, a large value for *max* possibly causing an overflow is needed so that the difference can be observed.

Another way to avoid race conditions is to use **atomic access** to variables as shown in Listing 3.13.

```
#include <stdio.h>

int main (int argc, char *argv[]) {
  int max; sscanf (argv[1], "%d", &max);
  int sum = 0;
  #pragma omp parallel for
    for (int i = 1; i <= max; i++)
      #pragma omp atomic
        sum = sum + i;
  printf ("%d\n", sum);
  return 0;
}
```

Listing 3.13: Summation of integers from a given interval using a atomic variable access — faster.

Although `sum` is a single variable shared by all threads in the team, the program computes the correct result as the `omp atomic` directive instructs the compiler to generate code where `sum = sum + i` update is performed as a single atomic operation (possibly using a hardware supported read-modify-write instructions).

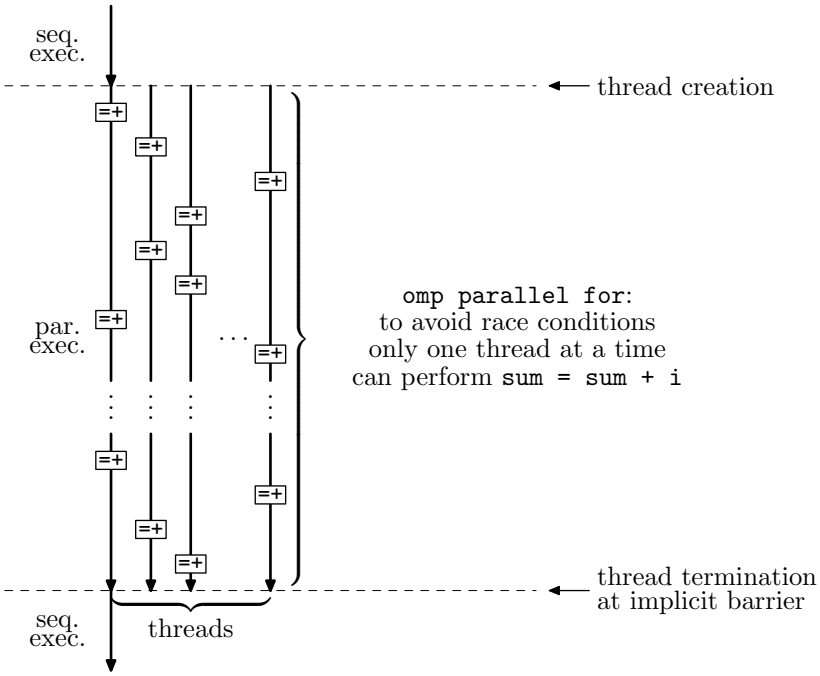


Fig. 3.10: Execution of the summation of integers as implemented in Listing 3.12.

The concepts of a critical section and atomic accesses to a variable are very similar, except that an atomic access is much simpler and thus usually faster than a critical section that can contain much more elaborate computation. Hence, the program in Listing 3.13 is essentially executed as illustrated in Figure 3.10.

OpenMP: critical sections

A critical section is decared as

`#pragma omp critical` *[(name)* `[hint(`*hint*`)]]`
structured-block

The *structured-block* is guaranteed to be executed by a single thread at a time. A critical section can be given *name*, an identifier with external linkage so that different tasks can implement their own implementation of the same critical section. A named critical section can be given a constant integer expression *hint* to establish a detailed control underlying locking.

To prevent race conditions and to avoid locking or explicit atomic access to variables at the same time, OpenMP provides a special operation called **reduction**. Using it, the program in Listing 3.11 is rewritten to the program shown in Listing 3.14.

```
1  #include <stdio.h>
2
3  int main (int argc, char *argv[]) {
4    int max; sscanf (argv[1], "%d", &max);
5    int sum = 0;
6    #pragma omp parallel for reduction(+:sum)
7      for (int n = 1; n <= max; n++)
8        sum = sum + n;
9    printf ("%d\n", sum);
10   return 0;
11 }
```

Listing 3.14: Summation of integers from a given interval using reduction — fast.

The additional clause `reduction(+:sum)` states that T private variables `sum` are created, one variable per thread. The computation within each thread is performed using the private variable `sum` and only when the parallel `for` loop has finished are the private variables `sum` added to variable `sum` declared in line 5 and printed out in line 9. The compiler and the OpemMP runtime system perform the final summation of local variables `sum` in a way suitable for the actual target architecture.

The program in Listing 3.14 is executed as shown in Figure 3.11. At first glance it is similar to the execution of the incorrect program in Listing 3.11, but there are no race conditions because in line 8 each thread uses its own private variable `sum`.

OpenMP: atomic access

Atomic access to a variable within *expression-stmt* is declared as

`#pragma omp atomic` *[*`seq_cst` *[*`,`*]] atomic-clause [[*`,`*]* `seq_cst`*]*
expression-stmt

or

`#pragma omp atomic` *[*`seq_cst`*]*
expression-stmt

when `update` is assumed. The `omp atomic` directive enforces an exclusive access to a storage location among all threads in the binding thread set without regard to the teams to which the threads belong.

The three most important *atomic-clause*s are the following:

- `read` causes an atomic read of `x` in statements of the form *expr* = `x`;
- `write` causes an atomic write to `x` in statements of the form `x` = *expr*;
- `update` causes an atomic update of `x` in statements of the form `++x`, `x++`, `--x`, `x--`, `x` = `x` *binop expr*, `x` = *expr binop* `x`, `x` *binop*= *expr*.

If `seq_cst` is used, an implicit flush operation of atomically accessed variable is performed after *statement-expr*.

OpenMP: reduction

Technically, reduction is yet another data sharing attribute specified by

`reduction(`*reduction-identifier*`:`*list*`)`

clause.

For each variable in the *list*, a private copy is created in each thread of a parallel region, and initialized to a value specified by the *reduction-identifier*. At the end of a parallel region, the original variable is updated with values of all private copies using the operation specified by the *reduction-identifier*.

The *reduction-identifier* may be `+`, `-`, `&`, `|`, `^`, `&&`, `||`, `min`, and `max`. For `*` and `&&` the initial value is `1`; for `min` and `max` the initial value is the largest and the smallest value of the variable's type, respectively; for all other operations, the initial value is `0`.

At the end of the parallel `for` loop, however, these private variables are added to the global variable `sum`.

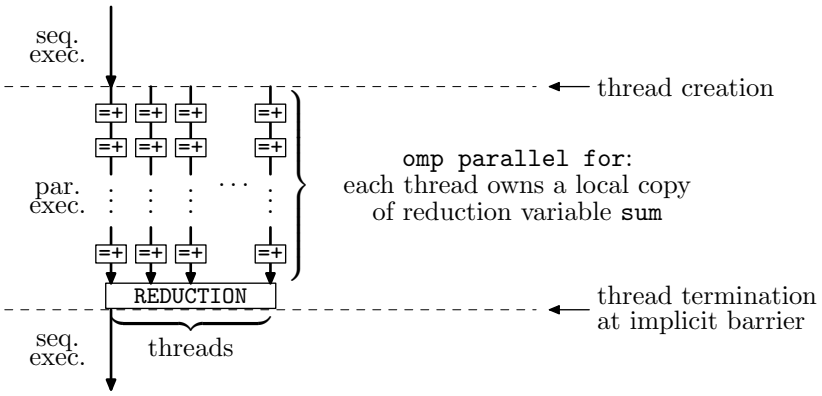


Fig. 3.11: Execution of the summation of integers as implemented in Listing 3.14.

Example 3.4. Computing π by numerical integration

There are many problems that require combining results of loop iterations. Let us start with a numerical integration in one dimension. Suppose we want to compute number π by computing the area of the unit circle defined by equation $x^2 + y^2 = 1$. Converting it to its explicit form $y = \sqrt{1-x^2}$, we want to compute number π using the equation

$$\pi = 4 \int_0^1 \sqrt{1-x^2}\,dx$$

as illustrated in Figure 3.12.

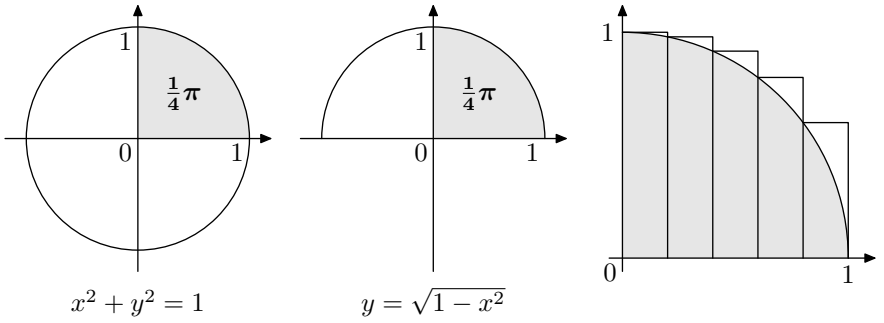


Fig. 3.12: Integrating $y = \sqrt{1-x^2}$ numerically from 0 to 1.

The integral on the right hand side of the equation above is computed numerically. Therefore, the interval $[0,1]$ is cut into N intervals $[\frac{1}{N}i, \frac{1}{N}(i+1)]$ where $0 \leq i \leq (N-1)$ for some chosen number of itervals N. To keep the program simple, the left Riemann sum is chosen: the area of each rectangle is computed as the width of a rectangle, i.e., $1/N$, multiplied by the function value computed in the left-end point of the interval, i.e., $\sqrt{1-(i/N)^2}$. Thus,

$$\int_0^1 \sqrt{1-x^2}dx \approx \sum_{i=0}^{N-1} \left(\frac{1}{N} \sqrt{1 - \left(\frac{1}{N} i \right)^2} \right)$$

for large enough N.

The program for computing π using the sum on the right hand side of the approximation is shown in Listing 3.15.

```
#include <stdio.h>
#include <math.h>

int main (int argc, char *argv[]) {
  int intervals; sscanf (argv[1], "%d", &intervals);
  double integral = 0.0;
  double dx = 1.0 / intervals;
  #pragma omp parallel for reduction(+:integral)
    for (int i = 0; i < intervals; i++) {
      double x = i * dx;
      double fx = sqrt (1.0 - x * x);
      integral = integral + fx * dx;
    }
  double pi = 4 * integral;
  printf ("%20.18lf\n", pi);
  return 0;
}
```

Listing 3.15: Computing π by integrating $\sqrt{1-x^2}$ from 0 to 1.

Despite the elements of numerical integration, the program in Listing 3.15 is remarkably similar to the program in Listing 3.14: after all, this numerical integra-

tion is nothing but a sumation of rectangle areas. Nevertheless, one important detail should not be missed: unlike `intervals`, `integral`, and `dx` variables `x` and `fx` must be thread private.

At this point it is perhaps worth showing once again that not every loop can be parallelized. By rewriting lines 7–13 of Listing 3.15 to code shown in Listing 3.16 a multiplication inside the loop was replaced by addition.

```
1  double x = 0.0;
2  #pragma omp parallel for reduction(+:integral)
3    for (int i = 0; i < intervals; i++) {
4      double fx = sqrt (1.0 - x * x);
5      integral = integral + fx * dx;
6      x = x + dx;
7    }
```

Listing 3.16: Computing π by integrating $\sqrt{1-x^2}$ from 0 to 1 using a non-paralellizable loop.

This works well if the program is run by only one thread (set `OMP_NUM_THREADS` to 1), but produces the wrong result if multiple threads are used. The reason is that the iterations are no longer independent: the value of `x` is propagated from one iteration to another so the next iteration cannot be performed until the previous has been finished. However, the `omp parallel for` directive in line 2 of Listing 3.16 states that the loop can and should be parallelized. The programmer unwisely requested the parallelization and took the responsibility for ensuring the loop can indeed be parallelized too lighty. □

Example 3.5. Computing π using random shooting

Another way of computing π is to shoot randomly into a square $[0,1]\times[0,1]$ and count how many shots hit inside the unit circle and how many do not. The ratio of hits vs. all shots is an approximation of the area of the unit circle within $[0,1]\times[0,1]$. As each shot is independent of another, shots can be distributed among different threads. Figure 3.13 illustrates this idea if two threads are used. The implementation, for any number of threads, is shown in Listing 3.17.

From the parallel programming view, the program in Listing 3.17 is basically simple: `num_shots` are shot within the parallel `for` loop in lines 17–23 and the number of hits is accumulated in variable `num_shots`. Furthermore, it also resembles the program in Listing 3.14: the results of independent iterations combined together to yield the final result.

The most intricate part of the program is generating random shots. The usual random generators, i.e., `rand` or `random`, are not *reentrant* or *thread-safe*: they should not be called in multiple threads because they use a single hidden state that is modified on each call regardless of the thread the call is made in. To avoid this problem, function `rnd` has been written: it takes a seed, modifies it and returns a random value in the interval $[0,1)$. Hence, a distinct seed for each thread is created in lines 12–14 where OpenMP function `omp_get_max_threads` is used to obtain the number of future threads that will be used in the parallel `for` loop later on. Using these seeds, the program contains one distinct random generator for each thread.

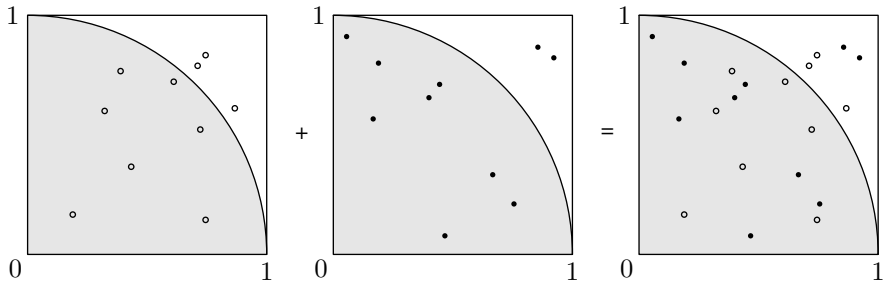


Fig. 3.13: Computing π by random shooting: different threads shoot independently, but the final result is a combination all shots.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <omp.h>
4
5  double rnd (unsigned int *seed) {
6    *seed = (1140671485 * (*seed) + 12820163) % (1 << 24);
7    return ((double)(*seed)) / (1 << 24);
8  }
9
10 int main (int argc, char *argv[]) {
11   int num_shots; sscanf (argv[1], "%d", &num_shots);
12   unsigned int seeds[omp_get_max_threads()];
13   for (int thread = 0; thread < omp_get_max_threads(); thread++)
14     seeds[thread] = thread;
15   int num_hits = 0;
16   #pragma omp parallel for reduction(+:num_hits)
17     for (int shot = 0; shot < num_shots; shot++) {
18       int thread = omp_get_thread_num();
19       double x = rnd (&seeds[thread]);
20       double y = rnd (&seeds[thread]);
21       if (x * x + y * y <= 1) num_hits = num_hits + 1;
22     }
23   double pi = 4.0 * (double)num_hits / (double)num_shots;
24   printf ("%20.18lf\n", pi);
25   return 0;
26 }
```

Listing 3.17: Computing π by random shooting using a parallel loop.

The rate of convergence towards π is much lower than if numerical integration is used. However, this example shows how simple it is to implement a wide class of Monte Carlo methods if random generator is applied correctly: one must only run all random based individual experiments, e.g., shots into $[0,1] \times [0,1]$ in lines 18–20, and aggregate the results, e.g., count the number of hits within the unit circle.

As long as the number of individual experiments is known in advance and the complexity of individual experiments is approximately the same, the approach in presented in Listing 3.17 suffices. Otherwise, a more sophisticated approach is needed, but more about that later.

Before proceeding, we can rewrite the program in Listing 3.17 to a simpler one. By splitting the `omp parallel` and `omp for` we can define the thread-local seed inside the parallel region as shown in Listing 3.19.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <omp.h>
4
5  double rnd (unsigned int *seed) {
6    *seed = (1140671485 * (*seed) + 12820163) % (1 << 24);
7    return ((double)(*seed)) / (1 << 24);
8  }
9
10 int main (int argc, char *argv[]) {
11   int num_shots; sscanf (argv[1], "%d", &num_shots);
12   int num_hits = 0;
13   #pragma omp parallel
14   {
15     unsigned int seed = omp_get_thread_num();
16     #pragma omp for reduction(+:num_hits)
17       for (int shot = 0; shot < num_shots; shot++) {
18         double x = rnd (&seed);
19         double y = rnd (&seed);
20         if (x * x + y * y <= 1) num_hits = num_hits + 1;
21       }
22   }
23   double pi = 4.0 * (double)num_hits / (double)num_shots;
24   printf ("%20.18lf\n", pi);
25   return 0;
26 }
```

Listing 3.18: Computing π by random shooting using a parallel loop.

Let us demonstrate that computing π by random shooting into $[0,1] \times [0,1]$ and counting the shots inside the unit circle can also be encoded differently as shown in Listing 3.19, but at its core it stays the same.

Namely, the parallel regions, one per each available thread, specified by the `omp parallel` directive in line 13 are used instead of the parallel `for` loop (see also Listings 3.1 and 3.2). Within each parallel region, the seed for the thread-local random generator is generated in lines 15. Then, the number of shots that must be carried out by the thread is computed in lines 16–18 and finally all shots are performed in a thread-local sequential `while` loop in lines 19–23. Unlike the iterations of the parallel `par` loop in Listing 3.18 the iterations of the `while` loop do not contain a call of function `omp_get_thread_num`. However, the aggregation of the results obtained by the parallel regions, i.e., the number of hits, is done using the reduction in the same way as in Listing 3.18. □

3.3.3 Distributing iterations among threads

So far no attention has been paid on how iterations of a parallel loop, or of a several collapsed parallel loops, are distributed among different threads in a single team of threads. However, OpenMP allows the programmer to specify several different iteration scheduling strategies.

Consider computing the sum of integers from a given interval again using the program shown in Listing 3.14. This time, however, the program will be modi-

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

double rnd (unsigned int *seed) {
  *seed = (1140671485 * (*seed) + 12820163) % (1 << 24);
  return ((double)(*seed)) / (1 << 24);
}

int main (int argc, char *argv[]) {
  int num_shots; sscanf (argv[1], "%d", &num_shots);
  int num_hits = 0;
  #pragma omp parallel reduction(+:num_hits)
    {
      unsigned int seed = omp_get_thread_num ();
      int loc_shots = (num_shots / omp_get_num_threads ()) +
        ((num_shots % omp_get_num_threads () > omp_get_thread_num ())
        ? 1 : 0);
      while (loc_shots-- > 0) {
        double x = rnd (seed);
        double y = rnd (seed);
        if (x * x + y * y <= 1) num_hits = num_hits + 1;
      }
    }
  double pi = 4.0 * (double)num_hits / (double)num_shots;
  printf ("%lf\n", pi);
  return 0;
}
```

Listing 3.19: Computing π by random shooting using parallel sections.

fied as shown in Listing 3.20. First, the `schedule(runtime)` clause is added to the `omp for` directive in line 8. It allows the iteration schedule strategy to be defined once the program is started using the shell variable `OMP_SCHEDULE`. Second, in line 10 each iteration prints out the the number of thread that executes it. And third, different iterations take different time to execute as specified by the argument of function `sleep` in line 11: iterations 1, 2, and 3 require 2, 3, and 4 seconds, respectively, while all other iterations require just 1 second.

```
#include <stdio.h>
#include <unistd.h>
#include <omp.h>

int main (int argc, char *argv[]) {
  int max; sscanf (argv[1], "%d", &max);
  long int sum = 0;
  #pragma omp parallel for reduction(+:sum) schedule(runtime)
    for (int i = 1; i <= max; i++) {
      printf ("%2d @ %d\n", i, omp_get_thread_num());
      sleep (i < 4 ? i + 1 : 1);
      sum = sum + i;
    }
  printf ("%ld\n", sum);
  return 0;
}
```

Listing 3.20: Summation of integers from a given interval where iteration scheduling strategy is determined in runtime.

OpenMP: scheduling parallel loop iterations

Distributing iterations of parallel loops among team threads is controlled by the `schedule` clause. The most important options are:

- `schedule(static)`: The iterations are divided into chunks each containing approximately the same number of iterations and each thread is given at most one chunk.
- `schedule(static,`*chunk_size*`)`: The iterations are divided into chunks where each chunk contains *chunk_size* iterations. Chunks are then assigned to threads in a round-robin fashion.
- `schedule(dynamic,`*chunk_size*`)`: The iterations are divided into chunks where each chunk contains *chunk_size* iterations. Chunks are assigned to threads dynamically: each thread takes one chunk at a time out of the common pool of chunks, executes it and requests a new chunk until the pool is empty.
- `schedule(auto)`: The selection of the scheduling strategy is left to the compiler and the runtime system.
- `schedule(runtime)`: The scheduling strategy is specified at run time using the shell variable `OMP_SCHEDULE`.

If no `schedule` clause is present in the `omp for` directive, the compiler and runtime system are allowed to choose the scheduling strategy on their own.

Suppose 4 threads are being used and $max = 14$:

- If `OMP_SCHEDULE=static`, the iterations are divided into chunks each containing approximately the same number of iterations and each thread is given at most one chunk. One possible `static` distribution of 14 iterations among 4 threads (but not the only one, see [18]) is

$$T_0: \underbrace{\{1,2,3,4\}}_{10\text{ secs}} \quad T_1: \underbrace{\{5,6,7,8\}}_{4\text{ secs}} \quad T_2: \underbrace{\{9,10,11\}}_{3\text{ secs}} \quad T_3: \underbrace{\{12,13,14\}}_{3\text{ secs}}$$

  Thread T_0 is assigned all the most time consuming iterations: although iteration when `i` equals 4 takes 1 second, iterations when `i` is either 1, 2, or 3 require 2, 3, and 4 seconds, respectivelly. Each iteration assigned to any other thread thakes only 1 second. Hence, thread T_0 finishes much later than all other threads as can be seen in Figure 3.14.
- If `OMP_SCHEDULE=static,1` or `OMP_SCHEDULE=static,2`, the iterations are divided into chunks containing 1 or 2 iterations, respectivelly. Chunks are then assigned to threads in a round-robin fashion as

$$T_0: \underbrace{\{1;5;9;13\}}_{5\text{ secs}} \quad T_1: \underbrace{\{2;6;10;14\}}_{6\text{ secs}} \quad T_2: \underbrace{\{3;7;11\}}_{6\text{ secs}} \quad T_3: \underbrace{\{4;8;12\}}_{3\text{ secs}}$$

  or

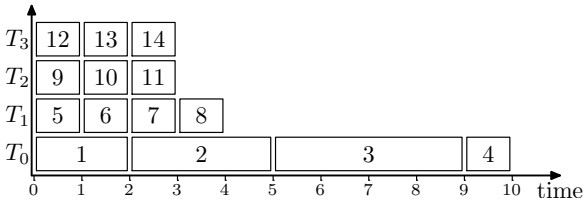


Fig. 3.14: A distribution of 14 iterations among 4 threads where iterations 1, 2 and 3 require more time than other iterations, using `static` iteration scheduling strategy.

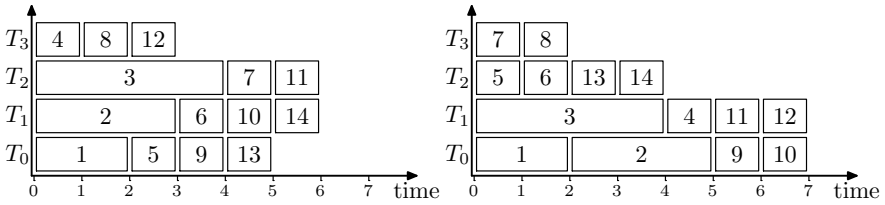


Fig. 3.15: A distribution of 14 iterations among 4 threads where iterations 1, 2 and 3 require more time than other iterations, using `static,1` (left) and `static,2` (right) iteration scheduling strategies.

$$T_0:\underbrace{\{1,2;9,10\}}_{7\text{ secs}} \quad T_1:\underbrace{\{3,4;11,12\}}_{7\text{ secs}} \quad T_2:\underbrace{\{5,6;13,14\}}_{4\text{ secs}} \quad T_3:\underbrace{\{7,8\}}_{2\text{ secs}}$$

where semicolon separates different chunks assigned to the same thread.
As shown in Figure 3.15 the running times of different threads differ less than if simple static scheduling is used. Furthermore, the overall running time is reduced from 10 seconds to 6 or 7 seconds, depending on the chunk size.

- If `OMP_SCHEDULE=dynamic,1` or `OMP_SCHEDULE=dynamic,2`, the iterations are divided into chunks containing 1 or 2 iterations, respectivelly. Chunks are assigned to threads dynamically: each thread takes one chunk at a time out of the common pool of chunks, executes it and requests a new chunk until the pool is empty. Hence, two possible dynamic assigments, for chunks consisting of 1 and 2 iterations, respectivelly, are

$$T_0:\underbrace{\{1;6;9;13\}}_{5\text{ secs}} \quad T_1:\underbrace{\{2;8;12\}}_{5\text{ secs}} \quad T_2:\underbrace{\{3;11\}}_{5\text{ secs}} \quad T_3:\underbrace{\{4;5;7;10;14\}}_{5\text{ secs}}$$

or

$$T_0:\underbrace{\{1,2\}}_{5\text{ secs}} \quad T_1:\underbrace{\{3,4\}}_{5\text{ secs}} \quad T_2:\underbrace{\{5,6;9,10;13,14\}}_{6\text{ secs}} \quad T_3:\underbrace{\{7,8;11,12\}}_{4\text{ secs}}$$

The scheduling of iterations is ilustrated in Figure 3.16: the overall running is further reduced to 5 or 6 seconds, again depending on the chunk size. The overall

running time of 5 seconds is the minimal possible as each thread performs the same amount of work.

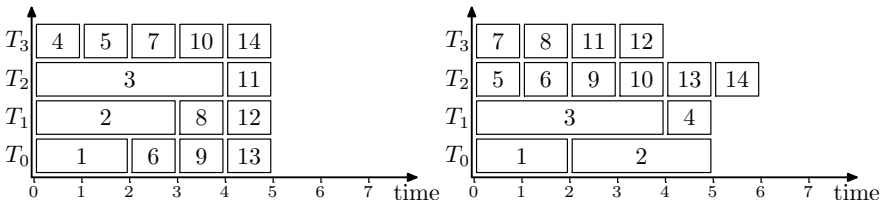


Fig. 3.16: A distribution of 14 iterations among 4 threads where iterations 1, 2 and 3 require more time than other iterations, using `dynamic,1` (left) and `dynamic,2` (right) iteration scheduling strategies.

Example 3.6. Mandelbrot set

The appropriate choice of iteration scheduling strategy always depends on the problem that is being solved. To see why the iteration scheduling strategy matters, consider computing the Mandelbrot set. It is defined as

$$M = \{c \; ; \; \limsup_{n\to\infty} |z_n| \leq 2 \text{ where } z_0 = 0 \; \wedge \; z_{n+1} = z_n^2 + c\}$$

and shown in Figure 3.17.

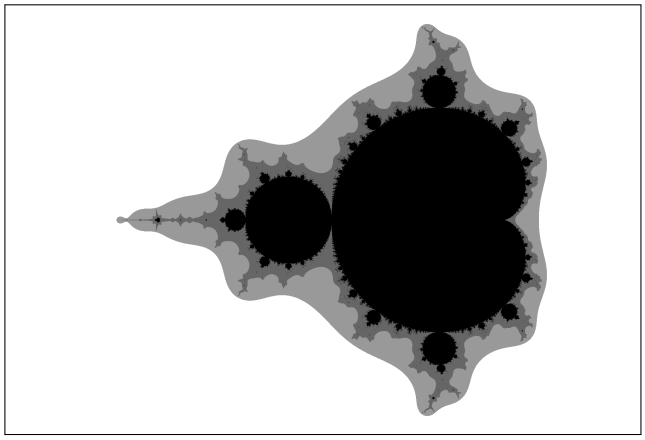


Fig. 3.17: The Mandelbrot set in the complex plane $[-2.6, +1.0] \times [-1.2i, +1.2i]$ (black) and its complement (white, light and dark gray). The darkness of each point indicates the time needed to establish whether a point belongs to the Mandelbrot set or not.

The Mandelbrot set can be computed using the program shown in Listing 3.21 which generates a picture consisting of `i_size`×`j_size` pixels. For each pixel the sequence $z_{n+1} = z_n^2 + c$ where c represents the pixel coordinates in the complex plane, is iterated until it either diverges ($|z_{n+1}| > 2$) or the maximum number of iterations (`max_iters`, defined in advance) is reached.

```
1  #pragma omp parallel for collapse(2) schedule(runtime)
2    for (int i = 0; i < i_size; i++) {
3      for (int j = 0; j < j_size; j++) {
4        // printf ("# (%d,%d) t=%d\n", i, j, omp_get_thread_num());
5        double c_re = min_re + i * d_re;
6        double c_im = min_im + j * d_im;
7
8        double z_re = 0.0;
9        double z_im = 0.0;
10       int iters = 0;
11       while ((z_re * z_re + z_im * z_im <= 4.0) &&
12              (iters < max_iters)) {
13         double new_z_re = z_re * z_re - z_im * z_im + c_re;
14         double new_z_im = 2 * z_re * z_im + c_im;
15         z_re = new_z_re; z_im = new_z_im;
16         iters = iters + 1;
17       }
18       picture[i][j] = iters;
19     }
20   }
```

Listing 3.21: Computing the Mandelbrot set.

To produce Figure 3.17, `max_iters` has been set to 100. Each point of the black region, i.e., within the Mandelbrot set, takes 100 iterations to compute. However, each point within the dark gray region requires more than 10 yet less than 100 iterations. Likewise, each point within the light gray region requires more than 5 and less than 10 iterations and all the rest, i.e., points colored white, require at most 5 iterations each. As different points and thus different iterations of the collapsed `for` loops in lines 2 and 3 require significantly different amount of computation, it matters what iteration scheduling strategy is used. Namely, if `static,100` is used instead of simply `static`, the running time is reduced by approximatelly 30 percent; the choice of `dynamic,100` reduces the running time even more. Run the program and measure its running time under different iteration scheduling strategies. □

3.3.4 The details of parallel loops and reductions

The parallel `for` loop and reduction operation are so important in OpenMP programming that they should be studied and understood in detail.

Let's return to the program for computing the sum of integers from 1 to *max* as shown in Listing 3.14. If it assumed that T, the number of threads, divides *max* and the `static` iteratin scheduling startegy is used, the program can be rewritten into the one shown in Listing 3.22. (See exercises for the case when T does not divide *max*.)

```
1  #include <stdio.h>
2  #include <omp.h>
3
4  int main (int argc, char *argv[]) {
5    int max; sscanf (argv[1], "%d", &max);
6    int ts = omp_get_max_threads ();
7    if (max % ts != 0) return 1;
8    int sums[ts];
9    #pragma omp parallel
10     {
11       int t = omp_get_thread_num ();
12       int lo = (max / ts) * (t + 0) + 1;
13       int hi = (max / ts) * (t + 1) + 0;
14       sums[t] = 0;
15       for (int i = lo; i <= hi; i++)
16         sums[t] = sums[t] + i;
17     }
18   int sum = 0;
19   for (int t = 0; t < ts; t++) sum = sum + sums[t];
20   printf ("%d\n", sum);
21   return 0;
22 }
```

Listing 3.22: Implementing efficient summation of integers by hand using simple reduction.

The initial thread first obtains T, the number of threads available (using OpenMP function `omp_get_max_threads`), and creates an array `sums` of variables used for summation within each thread. Although the array `sums` is going to be shared by all threads, each thread will access only one of its T elements.

Reaching `omp parallel` region the master thread creates $(T-1)$ slave threads to run alongside the master thread. Each thread, master or slave, first computes its subinterval (lines 11–12), initializes its local summation variable to 0 (line 13) and then executes its thread-local sequential `for` loop (line 14–15). Once all threads have finished computing local sums, only the master thread is left alive. It adds the local summation variables and prints the result. The overall execution is performed as shown in Figure 3.9. However, no race conditions appear becuase each thread uses its own summation variable, i.e., the `t`-th thread uses the `t`-th element `sums[t]` of array `sums`.

From the implementation point of view, the program in Listing 3.22 uses array `sums` instead of thread-local summation variables and performs the reduction by the master thread only. Array `sums` is created by the master thread before creating slave threads so that the explicit reduction, which is performed in line 18 after the slave threads have been terminated and their local variables (`t`, `lo`, `hi`, and `n`) have been lost, can be implemented.

Furtermore, the reduction is performed by adding local summation variables, i.e., elements of `sums`, one after another to variable `sum`. This takes $\mathrm{O}(T)$ time and works fine if the number of threads is small, e.g., $T = 4$ or $T = 8$. However, if there are a few hundred threads, a solution shown in Listing 3.23 that works in time $\mathrm{O}(\log_2 T)$ and produces the result in `sums[0]`, is often preferred (unless the target system architecture requires even more sophisticated method).

```
1     }
2   for (int d = 1; d < ts; d = d * 2)
3     #pragma omp parallel for
4       for (int t = 0; t < ts; t = t + 2 * d)
```

Listing 3.23: Implementing efficient summation of integers by hand using simple reduction.

The idea behind the code shown in Listing 3.23 is illustrated in Figure 3.18. In Listing 3.23 variable `d` contains the distance between elements of array `sums` being added, and as it doubles in each iteration, there are $\lceil \log_2 T \rceil$ iterations of the outer loop. Variable `t` denotes the left element of each pair being added in the inner loop. But as the inner loop is performed in parallel by at least $T/2$ threads which operate on distinct elements of array `sums`, all additions in the inner loop are performed simultaneously, i.e., in time $O(1)$.

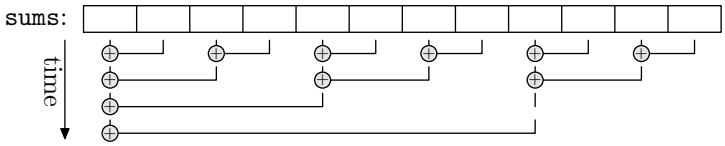


Fig. 3.18: Computing the reduction in time $O(\log_2 T)$ using $T/2$ threads when $T = 12$.

Note that either method used for computing the reduction uses $(T-1)$ additions. However, in the first method (line 18 of Listing 3.22) additions are performed one after another while in the second method (Listing 3.23) certain additions can be performed simultaneously.

3.4 Parallel tasks

Although most parallel programs spend most of their time running parallel loops, this is not always the case. Hence, it is worth exploring how a program consisting of different tasks can be parallelized.

3.4.1 Running independent tasks in parallel

As above where parallelization of loops that need not combine the results of its iterations was explained first, we start with explanation of **tasks** where cooperation is not needed.

Consider computing the sum of integers from 1 to *max* one more time. At the end of a previous section it was shown how iterations of a single parallel `for` loop are distributed among threads. This time, however, the interval from 1 to *max* is split into a number of mutually disjoint subintervals. For each subinterval a task that first computes the sum of all integers of a subinterval and then adds the sum of the subinterval to the global sum, is used.

The idea is implemented as the program in Listing 3.24. For the sake of simplicity, it is assumed that T, denoting the number of tasks and stored in variable `tasks`, divides *max*.

```
1  #include <stdio.h>
2  #include <omp.h>
3
4  int main (int argc, char *argv[]) {
5    int max; sscanf (argv[1], "%d", &max);
6    int tasks; sscanf (argv[2], "%d", &tasks);
7    if (max % tasks != 0) return 1;
8    int sum = 0;
9    #pragma omp parallel
10     {
11       #pragma omp single
12         for (int t = 0; t < tasks; t++) {
13           #pragma omp task
14             {
15               int local_sum = 0;
16               int lo = (max / tasks) * (t + 0) + 1;
17               int hi = (max / tasks) * (t + 1) + 0;
18               // printf ("%d: %d..%d\n", omp_get_thread_num(), lo, hi);
19               for (int i = lo; i <= hi; i++)
20                 local_sum = local_sum + i;
21               #pragma omp atomic
22                 sum = sum + local_sum;
23             }
24         }
25     }
26   printf ("%d\n", sum);
27   return 0;
28 }
```

Listing 3.24: Implementing summation of integers by using a fixed number of tasks.

Computing the sum is performed in the `parallel` block in lines 9–25. The `for` loop in line 12 creates all T tasks where each task is defined by the code in lines 13–23. Once the tasks are created, it is more or less up to OpenMP's runtime system to schedule tasks and execute them.

The important thing, however, is that the `for` loop in line 12 is executed by only one thread as otherwise each thread would create its own set of T tasks. This is achieved by placing the `for` loop in line 12 under the OpenMP directive `single`.

The OpenMP directive `task` in line 13 specifies that the code in lines 14–23 is to be executed as a single task. The local sum is initialized to 0 and the subinterval bounds are computed from the task number, i.e., `t`. The integers of the subinterval are added up and the local sum is added to the global sum using atomic section to prevent a race condition between two different tasks.

Note that when a new task is created, the execution of the task that created the new task continues without delay; once created, the new task has a life of its own. Namely, when the master thread in Listing 3.24 executes the `for` loop, it creates one new task in each iteration, but the iterations (and thus creation of new tasks) are executed one after another without waiting for the newly created tasks to finish (in fact, it would make no sense at all to wait for them to finish). However, all tasks must finish before the `parallel` region can end. Hence, once the global sum is printed out in line 26 of Listing 3.24, all tasks has already finished.

The difference between the approaches taken in the previous and this section can be told in yet another way. Namely, when iterations of a single parallel `for` loop are distributed among threads, tasks, one per thread, are created implicitly. But when a number of explicit tasks is used, the loop itself is split among tasks that are then distributed among threads.

Example 3.7. Fibonacci numbers

Computing the first *max* Fibonacci numbers using the time consuming recursive formula can be pretty naive, especially if a separate call of the recursive function is used for each of them. Nevertheless, it shows how to use the advantage of tasks.

The program in Listing 3.25 shows how this can be done. Note again that a single thread within a parallel region starts all tasks, one per each number. As the program is written, smaller Fibonacci numbers, i.e., for $n = 1, 2, \ldots$, are computed first while the largest are left to be computed later.

The time needed to compute the n-th Fibonacci number using function `fib` in line 4 of Listing 3.25 is of order $O(1.6^n)$. Hence, the the time complexity of in-

OpenMP: tasks

A task is declared using the directive

`#pragma omp task` *[clause [[,] clause] . . .]*
structured-block

The `task` directive creates a new task that executes *structured-block*. The new task can be executed immediately or can be deferred. A deferred task can be later executed by any thread in the team.

The `task` directive can be further refined by a number of clauses, the most important being the following ones:

- `final(`*scalar-logical-expression*`)` causes, if *scalar-logical-expression* evaluates to *true*, that the created task does not generate any new tasks any more, i.e., the code of would-be-generated new subtasks is included in and thus executed within this task;
- `if(`*[*`task:`*]**scalar-logical-expression*`)` causes, if *scalar-logical-expression* evaluates to *false*, that an undeferred task is created, i.e., the created task suspends the creating task until the created task is finished.

For other clauses see OpenMP specification.

OpenMP: limiting execution to a single thread

Within a `parallel` section, the directive

`#pragma omp single` *[clause [[,] clause] . . .]*
structured-block

causes *structured-block* to be executed by exactly one thread in a team (not necessarily the master thread). If not specified otherwise, all other threads wait idle at the implicit barrier at the end of the `single` directive.

The most important clauses are the following:

- `private(`*list*`)` specifies that each variable in the *list* is private to the code executed within the `single` dircetive;
- `nowait` removes the implicit barrier at the end of the `single` directive and thus allows other threads in the team to proceed without waiting for the code under the `single` directive to finish.

```
#include <stdio.h>
#include <omp.h>

long fib (int n) { return (n < 2 ? 1 : fib (n - 1) + fib (n - 2)); }

int main (int argc, char *argv[]) {
  int max; sscanf (argv[1], "%d", &max);
  #pragma omp parallel
    #pragma omp single
      for (int n = 1; n <= max; n++)
        #pragma omp task
          printf ("%d: %d %ld\n", omp_get_thread_num(), n, fib (n));
  return 0;
}
```

Listing 3.25: Computing Fibonacci numbers using OpenMP's tasks: smaller tasks, i.e., for smaller Fibonacci numbers are created first.

dividual tasks grows exponentially with n. Therefore it is perhaps better to create (and thus carry out) tasks in reverse order, the most demanding first and the least demanding last as shown in Listing 3.26. Run both programs, heck this hypothesis out and investigate which tasks get carried out by which thread. □

```
      for (int n = max; n >= 1; n--)
        #pragma omp task
          printf ("%d: %d\n", omp_get_thread_num(), n);
```

Listing 3.26: Computing Fibonacci numbers using OpenMP's tasks: smaller tasks, i.e., for smaller Fibonacci numbers are created last.

Converting a parallel `for` loop into a set of tasks is not very interesting and in most cases does not help either. The real power of tasks, however, can be appreciated when the number and the size of individual tasks cannot be known in advance. In

other words, when the problem or the algorithm demands that tasks are created dynamically.

Example 3.8. Quicksort

A good and simple, yet well-known example of this kind is sorting using the Quicksort algorithm [5]. The parallel version using tasks is shown in Listing 3.27.

```
1  void par_qsort (char **data, int lo, int hi,
2                  int (*compare)(const char *, const char*)) {
3    if (lo > hi) return;
4    int l = lo;
5    int h = hi;
6    char *p = data[(hi + lo) / 2];
7    while (l <= h) {
8      while (compare (data[l], p) < 0) l++;
9      while (compare (data[h], p) > 0) h--;
10     if (l <= h) {
11       char *tmp = data[l]; data[l] = data[h]; data[h] = tmp;
12       l++; h--;
13     }
14   }
15   #pragma omp task final(h - lo < 1000)
16     par_qsort (data, lo, h, compare);
17   #pragma omp task final(hi - l < 1000)
18     par_qsort (data, l, hi, compare);
19 }
```

Listing 3.27: The parallel implementation of the Quicksort algorithm where each recursive call is performed as a new task.

The partition part of the algorithm, implemented in lines 4–14 of Listing 3.27, is the same as in the sequential version. The recursive calls, thou, are modified because they can be performed independently, i.e., at the same time. Each of the two recursive calls is therefore executed as its own task.

However, no matter how efficient creating new tasks is, it takes time. Creating a new task only makes sense if a part of the table that must be sorted using a recursive call is big enough. In Listing 3.27, the clause `final` in lines 15 and 17 is used to prevent creating new tasks for parts of table that contain less that 1000 elements. The threshold 1000 has been chosen by experience; choosing the best threshold depends on many factors (the number of elements, the time needed to compare two elements, the implementation of OpenMP's tasks, ...). The experimental way of choosing it shall be, to some extend, covered in the forthcoming chapters.

There is an analogy with the sequential algorithm: recursion takes time as well and to speed up the sequential Quicksort algorithm, the insertion sort is used once the number of elements falls below a certain threshold.

There should be no confusion about the arguments for function `par_qsort`. However, function `par_qsort` must be called within a `parallel` region by exactly one thread as shown in Linsting 3.28.

As the Quicksort algorithm itself is rather efficient, i.e., it runs in time $O(n \log n)$, a sufficient number of elements must be used to see that the parallel version actually outperforms the sequential one. The comparison of runnning times is summarized in Table 3.1. By comparing the running times of the sequential version with the

```
1  #pragma omp parallel
2    #pragma omp single
3      par_qsort (strings, 0, num_strings - 1, compare);
```

Listing 3.28: The call of the parallel implementation of the Quicksort algorithm.

parallel version running within a single thread, one can estimate the time needed to create and destroy OpenMP's threads.

n	SEQ	PAR		
		(1 thread)	(4 threads)	(8 threads)
10^5	0.05 s	0.07 s	0.04 s	0.04 s
10^6	0.79 s	0.99 s	0.44 s	0.32 s
10^7	11.82 s	12.47 s	4.27 s	3.57 s
10^8	201.13 s	218.14 s	71.90 s	61.81 s

Table 3.1: The comparison of the running time of the sequential and parallel version of the Quicksort algorithm when sorting n random strings of max length 64 using a quad-core processor with multithreading.

Using 4 or 8 threads the parallel version is definitely faster, although the speed up is not proportional to the number of threads used. Note that the partition of the table in lines 4–14 of Listing 3.27 is performed sequentially and recall the Amdahl law. □

3.4.2 Combining the results of parallel tasks

In a number of cases parallel tasks cannot be left to execute independently of each other and leave its results in some global or shared variable. In such situation the programmer must take care of the life span of each individual task. The next example illustrates this.

Example 3.9. Quicksort revisited

Let us consider the Quicksort algorithm as an example again and modify it so that it returns the number of element pairs swapped during the partition phases.

Counting swaps during the partition phase in a sequential program is trivial. For instance, as shown in Listing 3.29 three new variables can be introduced, namely `count`, `locount` and `hicount`, that contain the number of swaps in the current partition phase and the total numbers of swaps in recursive calls, respectivelly. (In the sequential program this could be done with a single counter, but having three counters instead is more appropriate for the developing of the parallel version.)

In the parallel version the modification is not much harder, but a few things must be taken care of. First, as recursive calls in lines 16 and 18 of Listing 3.27 change

```
1  int par_qsort (char **data, int lo, int hi,
2                 int (*compare)(const char *, const char*)) {
3    if (lo > hi) return 0;
4    int l = lo;
5    int h = hi;
6    char *p = data[(hi + lo) / 2];
7    int count = 0;
8    while (l <= h) {
9      while (compare (data[l], p) < 0) l++;
10     while (compare (data[h], p) > 0) h--;
11     if (l <= h) {
12       count++;
13       char *tmp = data[l]; data[l] = data[h]; data[h] = tmp;
14       l++; h--;
15     }
16   }
17   int locount, hicount;
18   #pragma omp task shared(locount) final(h - lo < 1000)
19     locount = par_qsort (data, lo, h, compare);
20   #pragma omp task shared(hicount) final(hi - l < 1000)
21     hicount = par_qsort (data, l, hi, compare);
22   #pragma omp taskwait
23   return count + locount + hicount;
24 }
```

Listing 3.29: The call of the parallel implementation of the Quicksort algorithm.

to assignment statements in lines 19 in 21 of Listing 3.29, the values of variables `locount` and `hicount` are set in two newly created tasks and must therefore be shared among the creating and the created tasks. This is achieved using `shared` clause in lines 18 and 20.

Second, remember that once the new tasks in lines 18–19 and 20–21 are created, the task that has just created them continues. To prevent it from computing the sum of all three counters and returning the result when variables `locount` and `hicount` might not have been set yet, the `taskwait` directive is used. It represents an explicit barrier: all tasks created by the task executing it must finish before that task can continue.

At the end of the parallel section is an implicit barrier before all tasks created within the parallel section must finish just like all iterations of a parallel loop must. Hence, in Listing 3.24 there is no need for an explicit barrier using `taskwait`. □

OpenMP: explicit task barrier

An explicit task barrier is created by the following directive:

```
#pragma omp taskwait
```

It specifies a point in the program the task waits until all its subtasks are finished.

3.5 Exercises and mini projects

Exercises

1. Modify the program in Listing 3.1 so that it uses a team of 5 threads within the parallel region by default. Investigate how shell variables `OMP_NUM_THREADS` and `OMP_THREAD_LIMIT` influence the execution of the original and modified program.
2. If run with one thread per logical core, threads started by the program in Listings 3.1 print out their thread numbers in random order while threads started by the program in Listing 3.2 always print out their results in the same order. Explain why.
3. Suppose two 100×100 matrices are to be multiplied using 8 threads. How many dot products, i.e., operations performed by the innermost `for` loop, must each thread compute if different approaches to parallelizing the two outermost `for` loops of matrix multiplication illustrated in Figure 3.6 are used?
4. Draw a 3D graph with the size of the square matrix along one independent axis, e.g., from 1 to 100, and the number of available threads, e.g., from 1 to 16, along the other showing the ratio between the number of dot products computed by the most and the least loaded thread for different approaches to parallelizing the two outermost `for` loops of matrix multiplication illustrated in Figure 3.6.
5. Modify the programs for matrix multiplication based on different loop parallelization methods to compute $C = A \cdot B^{\mathrm{T}}$ instead of $C = A \cdot B$. Compare the running time of the original and modified programs.
6. Suppose 4 threads are being used when the program in Listing 3.20 and $max = 20$. Determine which iteration will be performed by which thread if `static,1`, `static,2` or `static,3` is used as a iteration scheduling strategy. Try without running the program first. (Assume that iterations 1, 2 and 3 require 2, 3 and 4 units of time while all other iterations require just 1 unit of time.)
7. Suppose 4 threads are being used when the program in Listing 3.20 and $max = 20$. Determine which iteration will be performed by which thread if `dynamic,1`, `dynamic,2` or `dynamic,3` is used as a iteration scheduling strategy. Is the solution uniquely defined? (Assume that iterations 1, 2 and 3 require 2, 3 and 4 units of time while all other iterations require just 1 unit of time.)
8. Modify lines 12 and 13 in Listing 3.22 so that the program works correctly even if T, the number of threads, does not divide max. The number of iterations of the `for` loop in lines 15 and 16 should not differ by more than 1 for any two threads.
9. Modify the program in Listing 3.22 so that the modified program implements `static,c` iteration scheduling strategy instead of `static` as is the case in Listing 3.22. The chunk size `c` must be a constant declared in the program.
10. Modify the program in Listing 3.22 so that the modified program implements `dynamic,c` iteration scheduling strategy instead of `static` as is the case in Listing 3.22. The chunk size `c` must be a constant declared in the program.

Hint: Use a shared counter of iterations that functions as a queue of not yet scheduled iterations outside the parallel section.

11. While computing the sum of all elements of `sums` in Listing 3.23, the program creates new threads within every iteration of the outer loop. Rewrite the code so that creation of new threads in every iteration of the outer loop is avoided.
12. Try rewriting the programs in Listings 3.25 and 3.26 using parallel `for` loops instead of OpenMP's tasks to mimic the behavior of the original program as close as possible. Find out which iteration scheduling strategy should be used. Compare the running time of programs using parallel `for` loops with those that use OpenMP's tasks.
13. Modify the program in Listing 3.27 so that it does not use `final` but works in the same way.
14. Check the OpenMP specification and rewrite the program in Listing 3.24 using the `taskloop` directive.

Mini projects

P1. Write a multicore program that uses CYK algorithm [13] to parse a string of symbols. The inputs are a context-free grammar G in Chomsky Normal Form and a string of symbols. At the end, the program should print YES if the string of symbols can be derived by the rules of the grammar and NO otherwise.
Write a sequential program (no OpenMP directives at all) as well. Compare the running time of the sequential program with the running time of the multicore program and compute the speedup for different grammars and different string lengths.

Hint: Observe that in the classical formulation of CYK algorithm the iterations of the outermost loop must be performed one after another but that iterations of the second outermost loop are independent and offer a good opportunity for parallelization.

P2. Write a multicore program for the "all-pairs shortest paths" problem [5]. The input is a weighted graph with no negative cycles and the expected output are lengths of the shortest paths between all pairs of vertices (where the length of a path is a sum of weights along the edges that the path consists of).
Write a sequential program (no OpenMP directives at all) as well. Compare the running time of the sequential program with the running time of the multicore program and compute the speedup achieved

1. for different number of cores and different number of threads per core, and
2. for different number of vertices and different number of edges.

Hint 1: Take the Bellman-Ford algorithm for all-pairs shortest paths [5] and consider its matrix multiplication formulation. For a graph $G = \langle V, E \rangle$ your program should achieve at least time $O(|V|^4)$, but you can do better and achieve time

$O(|V|^3 \log_2 |V|)$. In neither case should you ignore the cache performance: allocate matrices carefully.

Hint 2: Instead of using the Bellman-Ford algorithm, you can try parallelizing the Floyd-Warshall algorithm that runs in time $O(|V|^3)$ [5]. How fast is the program based on the Floyd-Warshall algorithm compared with the one that uses the $O(|V|^4)$ or $O(|V|^3 \log_2 |V|)$ Bellman-Ford algorithm?

3.6 Bibliographic notes

The primary source of information including all details of OpenMP API is available at OpenMP web site [20] where the complete specification [18] and a collection of examples [19] are available. OpenMP version 4.5 is used in this book as version 5.0 is still being worked on by OpenMP Architecture Review Board. The summary card for C/C++ is also available at OpenMP web site.

As standards and specifications are usually hard to read, one might consider some book wholly dedicated to programming using OpenMP. Although relatively old and thus lacking the most of the modern OpenMP features, the book by Rohit Chandra et al. [4] provides a nice introduction to underlying ideas upon which OpenMP is based upon and the basic OpenMP constructs. A more recent and comprehensive description of OpenMP, version 4.5, can be found in the book by Ruud van der Pas et al. [21].

Chapter 4
MPI processes and messaging

Abstract Distributed memory computers cannot communicate through a shared memory. Therefore, messages are used to coordinate parallel tasks that eventually run on geographically distributed but interconnected processors. Processes as well as their management and communication are well defined by a platform-independent message passing interface (MPI) specification. MPI is introduced from the practical point of view, with a set of basic operations that enable implementation of parallel programs. We will give simple example programs that will serve as an aid for a smooth start of using MPI and as motivation for developing more complex applications.

4.1 Distributed memory computers can execute in parallel

We know from previous chapters that there are two main differences between the shared memory and distributed memory computer architectures. The first difference is in the price of communication: the time needed to exchange a certain amount of data between two or more processors is in favor of shared memory computers, as these can usually communicate much faster than the distributed memory computers. The second difference, which is in the number of processors that can cooperate efficiently, is in favor of distributed memory computers. Usually, our primary choice when computing complex tasks will be to engage a large number of fastest available processors, but the communication among them poses additional limitations. Cooperation among processors implies communication or data exchange among them. When the number of processors must be high (e.g., more than eight) to reduce the execution time, the speed of communication becomes a crucial performance factor.

There is a significant difference in the speed of data movement between two computing cores within a single multicore computer, depending on the location of data to be communicated. This is because the data can be stored in registers, cache memory, or system memory, which can differ by up to two orders of magnitude if their access times are considered. The differences in the communication speed get

even more pronounced in the interconnected computers, again by orders of magnitude, but this now depends on the technology and topology of the interconnection networks and on the geographical distance of the cooperating computers.

Taking into account the above facts, complex tasks can be executed efficiently either (i) on a small number of extremely fast computers or (ii) on a large number of potentially slower interconnected computers. In this chapter, we focus on the presentation and usage of the **Message Passing Interface (MPI)**, which enables system-independent parallel programming. The well-established MPI standard[1] includes process creation and management, language bindings for C and Fortran, point-to-point and collective communications, and group and communicator concepts. Newer MPI standards are trying to better support the scalability in future extreme-scale computing systems, because currently, the only feasible option for increasing the computing power is in increased number of cooperating processors. Advanced topics, as one-sided communications, extended collective operations, process topologies, external interfaces, etc., are also covered by these standards, but are beyond the scope of this book.

The final goal of this chapter is advise users how to employ the basic MPI principles in the solution of complex problems with a large number of processes that exchange application data through messages.

4.2 Programmer's view

Programmers have to be aware that the cooperation among processes implies the data exchange. The total execution time is consequently a sum of computation and communication time. Algorithms with only local communication between neighboring processors are faster and more scalable than the algorithms with the global communication among all processors. Therefore, the programmer's view of a problem that will be parallelized has to incorporate a wide number of aspects, e.g. data independency, communication type and frequency, balancing the load among processors, balancing between communication and computation, overlapping communication and computation, synchronous or asynchronous program flow, stopping criteria, and others.

Most of the above issues that are related to communication are efficiently solved by the MPI specification. Therefore, we will identify the mentioned aspects and describe efficient solutions through the standardized MPI operations. Further sections should not be considered as an MPI reference guide or MPI library implementation manual. We will just try to rise the interest of readers, through simple and illustrative examples, and to show how some of the typical problems can be efficiently solved by the MPI methodology.

---

[1] Against potential ambiguities, some segments of text are reproduced from: A Message-Passing Interface Standard (Version 3.1), © 1993, 1994, 1995, 1996, 1997, 2008, 2009, 2012, 2015, by University of Tennessee, Knoxville, Tennessee.

4.3 Message passing interface

The standardization effort of a message passing interface (MPI) library began in nineties and is one of the most successful project of the software standardization. Its driving force was, from the beginning, a cooperation between academia and industry that has been created with the MPI standardization forum.

The MPI library interface is a specification, not an implementation. The MPI is not a language, and all MPI operations are expressed as functions, subroutines, or methods, according to the appropriate language bindings for C and Fortran, which are a part of the MPI standard. The MPI standard defines the syntax and semantics of library operations that support the message-passing model, independently of program language or compiler specification.

Since the word "`PARAMETER`" is a keyword in the Fortran language, the MPI standard uses the word "argument" to denote the arguments to a subroutine. It is expected that C programmers will understand the word "argument", which has no specific meaning in C, as a "parameter", thus allowing to avoid unnecessary confusion for Fortran programmers.

An MPI program consists of autonomous processes that are able to execute their own code in the sense of multiple instruction multiple data (MIMD) paradigm. An MPI process can be interpreted in this sense as a program counter that addresses their program instructions in the system memory, which implies that the program codes executed by each process have not to be the same.

The processes communicate via calls to MPI communication operations, independently of operating system. The MPI can be used in a wide range of programs written in C or Fortran. Based on the MPI library specifications several efficient MPI library implementations have been developed, either in open-source in the public domain. The success of the project is evidenced by a coherent development of the parallel software projects that are portable between different computing environments, e.g. parallel computers, clusters, and heterogeneous networks, and scalable along wide numbers of cooperating processors, from one to millions. Finally, the MPI interface is designed for end users, parallel library writers and developers of parallel software tools.

Any MPI program should have operations to initialize execution environment and to control starting and terminating procedures of all generated processes. MPI processes can be collected into groups of specific `size` that can communicate in its own environment where each message sent in a context must be received only in the same context. A **process group** and **context** together form an MPI `communicator`. A process is identified by its `rank` in the group associated with a communicator. There is a default communicator `MPI_COMM_WORLD` whose group encompass all initial processes, and whose context is default. Two essential questions arise early in any MPI parallel program: "How many processes are participating in computation?" and "Which are their identities?" Both questions will be answered after calling two specialized MPI operations.

The basic MPI communication is characterized by two fundamental MPI operations `MPI_SEND` and `MPI_RECV` that provide sends and receives of process data,

represented by numerous data types. Besides the data transfer these two operations synchronize the cooperating processes in time instants where communication has to be established, e.g. a process cannot proceed if the expected data has not arrived. Further, a sophisticated addressing is supported within a group of ranked processes that are a part of a communicator. A single program may use several communicators, which manage common or separated MPI processes. Such a concept enables to use different MPI based parallel libraries that can communicate independently, without interference, within a single parallel program.

Even that the most of parallel algorithms can be implemented by just a few MPI operations, the MPI-1 standard offers a set of more than 120 operations for elegant and efficient programming, including operations for collective and asynchronous communication in numerous topologies of interconnected computers. The MPI library is well documented from its beginning and constantly developing. The MPI-2 provides standardized process startup, dynamic process creation and management, improved data types, one-sided communication and versatile input/output operations. The MPI-3 standard introduces non blocking collective communication that enable communication-computation overlapping and the MPI Shared Memory (SHM) model that enables efficient programming of hybrid architectures, e.g. a network of multicore computing nodes.

Complete MPI is quite a large library with 128 MPI-1 operations, with twice as much in MPI-2 and even more in MPI-3. We will start with only six basic operations and further add a few from the complete MPI set for greater flexibility in the parallel programming. However, to fulfil the desires of this textbook one need to master just a few dozens of MPI operations that will be described in more details in the following sections.

Very well organized documentation can be found on several web-pages, for example on the following link: `http://www.mcs.anl.gov/research/projects/mpi/tutorial/mpiexmpl/contents.html` with assignments, solution, program output and many useful hints and additional links. The latest MPI standard and further information about MPI are available on `http://www.mpi-forum.org/`.

Example 4.1. Hello World MPI program

We will proceed with a minimal MPI program in C programming language. Its implementation is shown in Listing 4.1.

```
1  #include "stdafx.h"
2  #include <stdio.h>
3  #include "mpi.h"
4
5  int main(int argc, char **argv)
6  //int main(argc, argv)
7  //int  argc;
8  //char **argv;
9  {
10   int rank, size;
11   MPI_Init(&argc, &argv);
12   MPI_Comm_size(MPI_COMM_WORLD, &size);
13   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
14   printf("Hello world from process %d of %d processes.\n", rank, size);
15   MPI_Finalize();
```

```
16   return 0;
17 }
```

Listing 4.1: ”Hello world” MPI program `MSMPIHello.ccp` in C programming syntax.

The ”Hello World” has been written in C programming language, hence the three-line preamble should be commented and replaced by: `int main(int argc, char **argv)`, if C++ compiler is used. The ”Hello World” code looks like a standard C code with several additional lines with `MPI_` prefix, which are calls to global MPI operations that are executed on all processes. Note, that some MPI operations, that will be introduced later, could be local, i.e. executed on a single process.

The ”Hello World” code in Listing 4.1 is the same for all processes. It has to be compiled only once to be executed on all active processes. Such a methodology could simplify the development of parallel programs. Run the program with:

```
$ mpiexec -n 3 MSMPIHello
```

from Command prompt of the host process, at the path of directory where `MSMPIHello.exe` is located. The program should output three ”Hello World” messages, each with a process identification data.

All non-MPI procedures are local, e.g. `printf` in the above example. It runs on each process and prints separate ”Hello World” notice. If one would prefer to have only a notice from a specific process, e.g. `0`, an extra `if(rank == 0)` statement should be inserted. Let us comment the rest of the above code:

- `#include "stdafx.h"` is needed because the MS Visual Studio compiler has been used,
- `#include <stdio.h>` is needed because of `printf`, which is used later in the program,
- `#include "mpi.h"` provides basic MPI definition of named constants, types, and function prototypes, and must be included in any MPI program.

The above MPI program, including definition of variables, will be executed in all active processes. The number of processes will be determined by parameter `-n` of the MPI execution utility `mpiexec`, usually provided by the MPI library implementation.

- `MPI_Init` initializes the MPI execution environment and `MPI_Finalize` exits the MPI,
- `MPI_Comm_size(MPI_COMM_WORLD, &size)` returns `size`, which is the number of started processes, and
- `MPI_Comm_rank(MPI_COMM_WORLD, &rank)` that returns `rank`, i.e. an ID of each process.
- MPI operations return a status of the execution success; in C routines as the value of the function, which is not considered in the above C program, and in Fortran routines as the last argument of the function call (see Listing 4.2).

Depending on the number of processes, the `printf` function will run on each process, which will print a separate "Hello World" notice. If all processes will print the output, we expect `size` lines with "Hello World" notice, one from each process. Note that the order of the printed notices is not known in advance, because there is no guaranty about the ordering of the MPI processes. We will address this topic, in more details, later in this chapter. Note also that in this simple example no communication between processes has been required.

□

For comparison, a version of "Hello World" MPI program in Fortran programming language is given in Listing 4.2:

```
1  program hello_world
2  include '/usr/include/mpif.h'
3  integer ierr, num_procs, my_id
4
5  call MPI_INIT (ierr)
6  call MPI_COMM_RANK (MPI_COMM_WORLD, my_id, ierr)
7  call MPI_COMM_SIZE (MPI_COMM_WORLD, num_procs, ierr)
8  print *, "Hello world from process ", my_id, " of ", num_procs
9  call MPI_FINALIZE (ierr)
10 stop
11 end
```

Listing 4.2: "Hello world" MPI program `OMPIHello.f` in Fortran programming language.

Note that capitalized `MPI_` prefix is used again in the names of MPI operations, which are also capitalized in Fortran syntax, but the different header file `mpif.h` is included. MPI operations return a status of execution success, i.e. `ierr` in the case of Fortran program.

4.3.1 MPI operation syntax

The MPI standard is independent of specific programming languages. To stress this fact capitalized MPI operation names will be used in the definition of MPI operations. MPI operation arguments, in a language-independent notation, are marked as:

IN - for input values that may be used by the operation, but not updated;

OUT - for output values that may be updated by the operation, but not used as input value;

INOUT - for arguments that may be used and/or updated by the MPI operation. An argument, used as IN by some processes and as OUT by other processes is also marked as INOUT, even that it is not used for input and for output in a single call.

For shorter specifications of MPI operations, the following notation is used for descriptive names of arguments:

IN arguments are in normal text, e.g. `buf`, `sendbuf`, `MPI_COMM_WORLD`, etc.

OUT arguments are in underlined text, e.g. rank, recbuf, etc.

INOUT arguments are in underlined italic text, e.g *inbuf*, *request*, etc.

The examples of MPI programs, in the rest of this chapter, are given in C programming language. Below are some terms and conventions that are implemented with C program language binding:

- Function names are equal to the MPI definitions but with the `MPI_` prefix and the first letter of the function name in upper case, e.g. `MPI_Finalize()`.
- The status of execution success of MPI operations is returned as integer return codes, e.g. `ierr = MPI_Finalize()`. The return code can be an error code or `MPI_SUCCESS` for successful competition, defined in the file `mpi.h`, Note that all predefined constants and types are fully capitalized.
- Operation arguments IN are usually passed by value with an exception of the send buffer, which is determined by its initial address. All OUT and INOUT arguments are passed by reference (as pointers), e.g.
  `MPI_Comm_size (MPI_COMM_WORLD, &size)`.

4.3.2 MPI data types

MPI communication operations specify the message data length in terms of number of data elements, not in terms of number of bytes. Specifying message data elements is machine independent and closer to the application level. In order to retain machine independent code, the MPI standard defines its own basic data types that can be used for the specification of message data values, and correspond to the basic data types of the host language.

As MPI does not require that communicating processes use the same representation of data, i.e. data types, it needs to keep track of possible data types through the build-in basic MPI data types. For more specific applications, MPI offers operations to construct custom data types, e.g. array of `(int, float)` pairs, and many other options. Even that the type casting between a particular language and the MPI library may represent a significant overhead, the portability of MPI programs significantly benefits.

Some basic MPI data types, that correspond to the adequate C or Fortran data types, are listed in Table 4.1. Details on advanced structured and custom data types can be found in the before mentioned references.

The data types `MPI_BYTE` and `MPI_PACKED` do not correspond to a C or a Fortran data type. A value of type `MPI_BYTE` consists of a byte, i.e. 8 binary digits. A byte is uninterpreted and is different from a character. Different machines may have different representations for characters, or may use more than one byte to represent characters. On the other hand, a byte has the same binary value on all machines. If the size and representation of data is known, the fastest way is the transmission of raw data, for example, by using an elementary MPI data type `MPI_BYTE`.

Table 4.1: Some MPI data types corresponding to C and Fortran data types.

MPI data type	C data type	MPI data type	Fortran data type
MPI_INT	int	MPI_INTEGER	INTEGER
MPI_SHORT	short int	MPI_REAL	REAL
MPI_LONG	long int	MPI_DOUBLE_PRECISION	DOUBLE PRECISION
MPI_FLOAT	float	MPI_COMPLEX	COMPLEX
MPI_DOUBLE	double	MPI_LOGICAL	LOGICAL
MPI_CHAR	char	MPI_CHARACTER	CHARACTER
MPI_BYTE	/	MPI_BYTE	/
MPI_PACKED	/	MPI_PACKED	/

The MPI communication operations have involved only buffers containing a continuous sequence of identical basic data types. Often, one wants to pass messages that contain values with different data types, e.g. a number of integers followed by a sequence of real numbers; or one wants to send non-contiguous data, e.g. a sub-block of a matrix. The type `MPI_PACKED` is maintained by `MPI_PACK` or `MPI_UNPACK` operations, which enable to pack different types of data into a contiguous send buffer and to unpack it from a contiguous receive buffer.

A more efficient alternative is a usage of derived data types for construction of custom message data. The derived data types allow, in most cases, to avoid explicit packing and unpacking, which requires less memory and time. A user specifies in advance the layout of data types to be sent or received and the communication library can directly accesses a non-continuous data. The simplest non-contiguous datatype is the `vector` type, constructed with `MPI_Type_vector`. For example, a sender process has to communicate the main diagonal of an $N \times N$ array of integers, declared as:

```
int matrix[N][N];
```

which is stored in a row-major layout. A continuous derived datatype `diagonal` can be constructed:

```
MPI_Datatype MPI_diagonal;
```

that specifies the main diagonal as a set of integers:

```
MPI_Type_vector (N, 1, N+1, MPI_INT, &diagonal);
```

where their count is `N`, block length is `1`, and stride is `N+1`. The receiver process receives the data as a contiguous block. There are further options that enable the construction of sub-arrays, structured data, irregularly strided data, etc.

If all data of an MPI program is specified by MPI types it will support data transfer between processes on computers with different memory organization and different interpretations of elementary data items, e.g. in heterogeneous platforms. The parallel programs, designed with MPI data types, can be easily ported even between computers with unknown representations of data. Further, the custom application-

oriented data types can reduce the number of memory-to-memory copies or can be tailored to a dedicated hardware for global communication.

4.3.3 MPI error handling

The MPI standard assumes a reliable and error free underlying communication platform therefore, it does not provide mechanisms for dealing with failures in the communication system. For example, a message sent is always received correctly, and the user need not check for transmission errors, time-outs, or similar. Similarly, MPI does not provide mechanisms for handling processor failures. A program error can follow an MPI operation call with incorrect arguments, e.g. non-existing destination in a send operation, exceeding available system resources, or similar.

Most of MPI operation calls return an error code that indicates the completion status of the operation. Before the error value is returned, the current MPI error handler is called, which, by default, aborts all MPI processes. However, MPI provides mechanisms for users to change this default and to handle recoverable errors. One can specify that no MPI error is fatal, and handle the returned error codes by custom error-handling routines.

4.3.4 Make your computer ready for using MPI

In order to test the presented theory we need to install first the necessary software that will make our computer ready for running and testing MPI programs. In Appendix A of this book, readers will find short instructions for the installation of free MPI supporting software for either for Linux, macOS or MS Windows powered computers. Beside a compiler for selected program language, an MPI implementation of the MPI standard is needed with a method for running MPI programs. Please, refer the instruction in Appendix A and run your first "Hello Word" MPI program. Then you can proceed here in order to find some further hints for running and testing simple MPI programs, either on a single multicore computer or on a set of interconnected computers.

4.3.5 Running and configuring MPI processes

Any MPI library will provide you with the `mpiexec` (or `mpirun`) program that can launch one or more MPI applications on a single computer or on a set of interconnected computers (hosts). The program has many options that are standardized to some extent, but one is advised to check actual program options with `mpiexec -help`. Most common options are `-n <num_processes>`, `-host` or `-machinefile`.

An MPI program executable `MyMPIprogram.exe` can be launched on a local host and on three processes with:

```
$ mpiexec -n 3 MyMPIprogram
```

MPI will automatically distribute processes among the available core, which can be specified by option `-cores <num_cores_per_host>` Alternatively, the program can be launched on two interconnected computers, on each with four processes, with:

```
$ mpiexec -host 2 host1 4 host2 4 MyMPIprogram
```

For more complex managing of cooperation processes, a separate configuration file can be used. The processes available for the MPI can be specified by using `-machinefile` option to `mpiexec`. With this option, a text file, e.g. `myhostsfile`, lists computers on which to launch MPI processes. The hosts are listed one per line, identified either with a computer name or with its IP address. An MPI program, e.g. `MyMPIprogram`, can be executed, for example on three processes, with:

```
$ mpiexec -machinefile myhostsfile -n 3 MyMPIprogram
```

Single computer

The configuration file can be used for a specification of processes on a single computer or on a set of interconnected computers. For each host, the number of processes to be used on that host can be defined by a number that follows a computer name. For example, on a computer with a single core, the following configuration file defines four processes per computing core:

```
localhost 4
```

If your computer has, for example, four computing cores, MPI processes will be distributed among the cores automatically, or in a way specified by the user in the MPI configuration file, which supports, in this case, the execution of the MPI parallel program on a single computer. The configuration file could have the following structure:

```
localhost
localhost
localhost
localhost
```

specifying that a single process will run on each computing core if `mpiexec` option `-n 4` is used, or two processes will run on each computing core if `-n 8` is used, etc. Note, that there are further alternative options for configuring MPI processes that are usually described in more details in `-help` options of a specific MPI implementation.

Your computer is now ready for the coding and testing more useful MPI programs that will be discussed in following sections. Before that, some further hints are given for the execution of MPI programs on a set of interconnected computers.

Interconnected computers

If you are testing your program on a computer network you may select several computers to perform defined processes and run and test your code. The configuration file must be edited in a way that all cooperating computers are listed. Suppose that four computers will cooperate, each with two computing cores. The configuration file: `myhostsfile` should contain names or IP addresses of these computers, e.g.:

```
computer_name1
computer_name2
192.20.301.77
computer_name4
```

each in a separate line, and with the first name belonging to the name of the local host, i.e. the computer from which the MPI program will be started, by `mpiexec`.

Let us execute our MPI program `MyMPIprogram` on a set of computers in a network, e.g. connected with an Ethernet. Editing, compiling and linking process is the same as in the case of a single computer. However, the MPI executable should available to all computers, e.g. by a manual copying of the MPI executable on the same path on all computers, or more systematically, through a shared disk.

On MS Windows, a service for managing the local MPI processes, e.g. smpd daemons should be started by: `smpd -d` on all cooperating computers before launching MPI programs. The cooperating computers should have the same version of the MPI library installed, and the compiled MPI executable should be compatible with the computing platforms (32 or 64 bits) on all computers. The command from the master host:

```
$mpiexec -machinefile myhostsfile \\MasterHost\share\MyMPIprog
```

will enable to run the program on a set of processes, eventually located on different computers, as has been specified in the configuration file `myhostsfile`.

Note also, that the a potential user should be granted with rights for executing the programs on selected computers. One will need a basic user account and an access to the MPI executable that must be located on the same path on all computers. In Linux, this can be accomplished automatically by placing the executable in `/home/username/` directory. Finally, a method that allow automatic login, e.g. in Linux, SSH login without password, is needed, to enable automatic login between cooperating computers.

The described approach is independent on the technology of the interconnection network. The interconnected computers can be multicore computers, computing clusters connected by Gigabit Ethernet or Infiniband, or computers in a home network connected by Wi-Fi.

4.4 Basic MPI operations

Let us recall the presented issues in a more systematic way by a brief description of four basic MPI operations. Two trivial operations without MPI arguments will initiate and shut down the MPI environment. Next two operations will answer the questions: "How many processes will cooperate?" and "Which is my ID among them?" Note that all four operations are called from all processes of the current communicator.

4.4.1 `MPI_INIT (int *argc, char ***argv)`

The operation initiates an MPI library and environment. The arguments `argc` and `argv` are required in C language binding only, where they are parameters of the main C program.

4.4.2 `MPI_FINALIZE ()`

The operation shuts down the MPI environment. No MPI routine can be called before `MPI_INIT` or after `MPI_FINALIZE`, with one exception `MPI_INITIALIZED (flag)`, which queries if `MPI_INIT` has been called.

4.4.3 `MPI_COMM_SIZE (comm, size)`

The operation determines the number of processes in the current communicator. The input argument `comm` is the handle of communicator; the output argument `size` returned by the operation `MPI_COMM_SIZE` is the number of processes in the group of `comm`. If `comm` is `MPI_COMM_WORLD` then it represents the number of all active MPI processes.

4.4.4 `MPI_COMM_RANK (comm, rank)`

The operation determines the identifier of the current process within a communicator. The input argument `comm` is the handle of the communicator; the output argument `rank` is an ID of the process from `comm`, which is in the range from `0` to `size-1`.

In the following sections, some of the frequently used communication MPI operations are described briefly. There is no intention to provide an MPI user manual in its complete version, instead, this short description should be just a first motivation for beginners to write an MPI program that will effectively harness his computer, and to further explore the beauty and usefulness of the MPI approach.

4.5 Process-to-process communication

We know from previous chapters that a traditional process is associated with a private program counter of its private address space. Processes may have multiple program threads, associated with separate program counters, which share a single process's address space. The message passing model formalizes the communication between processes that have separate address spaces. The process-to-process communication has to implement two essential tasks: data movement and synchronization of processes, therefore it requires cooperation of sender and receiver processes. Consequently, every send operation expects a pairing/matching receive operation. The cooperation is not always apparent in the program, which may hinder the understanding of the MPI code.

A schematic presentation of a communication between sender `Process_0` and receiver `Process_1` is shown in Fig. 4.1. In this case optional intermediate message buffers are used in order to enable sender `Process_0` to continue immediately after it initiates the send operation. However, `Process_0` will have to wait on the return from the previous call, before it can send a new message. On the receiver side, `Process_1` can do some useful work instead of idling while waiting on the matching message reception. It is a communication system that must ensure that the message will be reliably transferred between both processes. If the processes have been created on a single computer, the actual communication will be probably implemented through a shared memory. If the processes reside on two distant computers, then the actual communication might be performed through an existing interconnection network using, e.g. TCP/IP communication protocol.

Although that blocking send/receive operations enable a simple way for synchronization of processes, they could introduce unnecessary delays in cases where sender and receiver do not reach communication point at the same real time. For example, if `Process_0` issues a send call significantly before the matching receive call in `Process_1`, `Process_0` will start waiting to the actual message data transfer. In the same way, processes' idling can happen if a process that produces many messages is much faster than the consumer process. Message buffering may alleviates the idling to some extent, but if the amount of data exceeds the capacity of the message buffer, which can always happen, `Process_0` will be blocked again.

The next concern of the blocking communication are deadlocks. For example, if `Process_0` and `Process_1` initiate their send calls in the same time, they will be blocked forever by waiting a matching receive calls. Fortunately, there are several

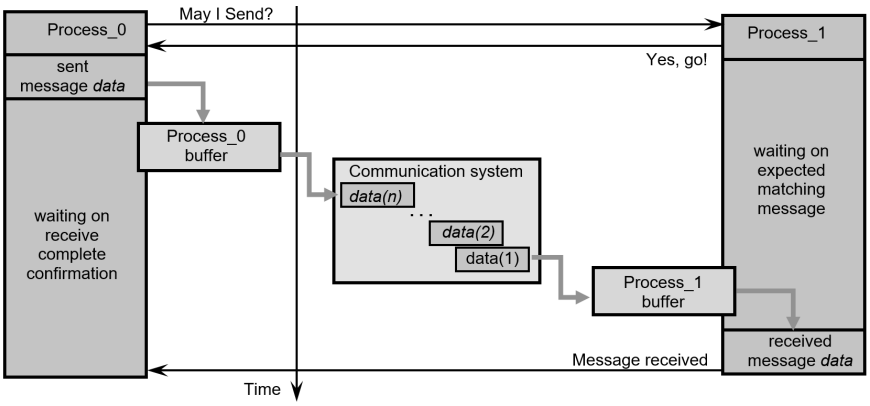


Fig. 4.1: Communication between two processes awakes both of them while transferring data from sender `Process_0` to receiver `Process_1`, possibly with a set of shorter sub-messages.

ways for alleviating such situations, which will be described in more details near the end of Section 4.7.

Before an actual process-to-process transfer of data happens, several issues have to be specified, e.g. how will message data be described, how processes will be identified, how the receiver recognizes/screens messages, when the operations will complete. The `MPI_SEND` and `MPI_RECV` operations are responsible for the implementation of the above issues.

4.5.1 `MPI_SEND (buf, count, datatype, dest, tag, comm)`

The operation, invoked by a blocking call `MPI_SEND` in the sender process `source`, will not complete until there is a matching `MPI_RECV` in receiver process `dest`, identified by a corresponding `rank`. The `MPI_RECV` will empty the input send buffer `buf` of matching `MPI_SEND`. The `MPI_SEND` will return when the message data has been delivered to the communication system and the send buffer `buf` of the sender process `source` can be reused. The send buffer is specified by the following arguments: `buf` - pointer to the send buffer, `count` - number of data items, and `datatype` - type of data items. The receiver process is addressed by an envelope that consists of arguments `dest`, which is the rank of receiver process within all processes in the communicator `comm`, and of a message `tag`.

The **message tags** provide a mechanism for distinguishing between different messages for the same receiver process identified by destination `rank`. The `tag` is an integer in the range `[0, UB]` where `UB`, defined in `mpi.h`, can be found by querying the predefined constant `MPI_TAG_UB`. When a sender process has to send more separate messages to a receiver process, the sender process will distinguish

them by using tags, which will allow receiver process to efficiently screening its messages. For example, if a receiver process has to distinguish between messages from a single source process, a message tag will serve an additional means for messages differentiation. MPI_ANY_TAG is a constant pre-defined in mpi.h, which can be considered as a "wild-card", where all tags will be accepted.

4.5.2 MPI_RECV (buf, count, datatype, source, tag, comm, status)

This operation waits until the communication system delivers a message with matching datatype, source, tag, and comm. Messages are screened at the receiving part based on specific source, which is a rank of the sender process within communicator comm, or not screened at all on source by equating it with MPI_ANY_SOURCE. The same screening is performed with tag, or if screening on tag is not necessary, by using MPI_ANY_TAG, instead. After return from MPI_RECV the output buffer buf is emptied and can be reused.

The number of received data items of datatype must be equal of fewer as specified by count, which must be positive or zero. Receiving more data items results an error. In such cases, the output argument status contains further information about the error. The entire set of arguments: count, datatype, source, tag and comm, must match between the sender process and the receiver process to initiate actual message passing. When a message, posted by a sender process, has been collected by a receiver process, the message is said to be completed, and the program flows of the receiver and the sender processes may continue.

Most implementations of the MPI libraries copy the message data out of the user buffer, which was specified in the MPI program, into some other intermittent system or network buffer. When the user buffer can be reused by the application, the call to MPI_SEND will return. This may happen before the matching MPI_RECV is called or it may not, depending on the message data length.

Example 4.2. Ping-pong message transfer

Let us check the behaviour of the MPI_SEND and MPI_RECV operations on your computer, if the message length grows. Two processes will exchange messages that will become longer and longer. Each process will report when the expected message has been sent, which means that it was also received. The code of the MPI program MSMPImessage.cpp is shown in Listing 4.3. Note, that there is a single program for both processes. A first part of the program, that determines the number of cooperating processes, is executed on both processes, which must be two. Then the program splits in two parts, first for process of rank = 0 and second of process of rank = 1. Each process sends and receives a message with appropriate calls to the MPI operations. We will see in the following, how the order of these two calls impacts the program execution.

```
#include "stdafx.h"
#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"
int main(int argc, char* argv[])
{
  int   numprocs, rank, tag = 100, msg_size=64;
  char  *buf;
  MPI_Status status;
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
  if (numprocs != 2) {
    printf("The number of processes must be two!\n");
    MPI_Finalize();
    return(0);
  }
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  printf("MPI process %d started...\n", rank);
  fflush(stdout);
  while (msg_size  < 10000000) {
    msg_size = msg_size *2;
    buf = (char *)malloc(msg_size * sizeof(char));
    if (rank == 0) {
      MPI_Send(buf, msg_size, MPI_BYTE, rank+1, tag, MPI_COMM_WORLD);
      printf("Message of length %d to process %d\n",msg_size,rank+1);
      fflush(stdout);
      MPI_Recv(buf, msg_size, MPI_BYTE, rank+1, tag,MPI_COMM_WORLD,
               &status);
    }
    if (rank == 1) {
//      MPI_Recv(buf, msg_size, MPI_BYTE, rank-1, tag, MPI_COMM_WORLD,
//               &status);
      MPI_Send(buf, msg_size, MPI_BYTE, rank-1, tag, MPI_COMM_WORLD);
      printf("Message of length %d to process %d\n",msg_size,rank-1);
      fflush(stdout);
      MPI_Recv(buf, msg_size, MPI_BYTE, rank-1, tag, MPI_COMM_WORLD,
               &status);
    }
    free(buf);
  }
  MPI_Finalize();
}
```

Listing 4.3: Verification of `MPI_SEND` and `MPI_RECV` operations on your computer.

The output of this program should be as follows:

```
$ mpiexec -n 2 MPImessage
MPI process 0 started...
MPI process 1 started...
Message of length 128 send to process 1.
Message of length 128 returned to process 0.
Message of length 256 send to process 1.
Message of length 256 returned to process 0.
Message of length 512 send to process 1.
Message of length 512 returned to process 0.
Message of length 1024 send to process 1.
Message of length 1024 returned to process 0.
Message of length 2048 send to process 1.
Message of length 2048 returned to process 0.
```

```
Message of length 4096 returned to process 0.
Message of length 4096 send to process 1.
Message of length 8192 returned to process 0.
Message of length 8192 send to process 1.
Message of length 16384 send to process 1.
Message of length 16384 returned to process 0.
Message of length 32768 send to process 1.
Message of length 32768 returned to process 0.
Message of length 65536 send to process 1.
Message of length 65536 returned to process 0.
```

The program blocks at the message length 65536, which is in some relation with the capacity of the MPI data buffer in the actual MPI implementation. When the message exceed it, `MPI_Send` in both processes block and enter a deadlock. If we just change the order of `MPI_Send` and `MPI_Recv` by comment lines 36-37 and uncomment lines 31-32 in process with `rank = 1`, all expected messages until the length 16777216 are transferred correctly. Some further discussion about the reasons for such a behaviour will be provided later, in Section 4.7. □

4.5.3 MPI_SENDRECV (sendbuf, sendcount, sendtype, dest, sendtag, recvbuf, recvcount, recvtype, source, recvtag, comm, status)

The MPI standard specifies several additional operations for message transfer that are a combination of basic MPI operations. They are useful for writing more compact programs. For example, operation `MPI_SENDRECV` combines a sending of message to destination process `dest` and a receiving of another message from process `source`, in a single call in sender and receiver process; however, with two distinct message buffers: `sendbuf`, which acts as an input, and `recvbuf`, which is an output. Note, that buffers' sizes and types of data can be different.

The send-receive operation is particularly effective for executing a shift operation across a chain of processes. If blocking `MPI_SEND` and `MPI_RECV` are used, then one needs to order them correctly, for example - even processes send, then receive, odd processes receive first, then send - so as to prevent cyclic dependencies that may lead to deadlocks. By using `MPI_SENDRECV` the communication subsystem will manage these issues alone.

There are further advanced communication operations that are a composition of basic MPI operations. For example `MPI_SENDRECV_REPLACE (buf, count, datatype, dest, sendtag, source, recvtag, comm, status)` operation implements the functionality the `MPI_SENDRECV`, but uses only a single message buffer. The operation is therefore useful in cases with send and receive messages of the same length and of the same data type.

Seven basic MPI operations

Many parallel programs can be written and evaluated just by using the following seven MPI operations that have been overviewed in the previous sections:

```
MPI_INIT,
MPI_FINALIZE,
MPI_COMM_SIZE,
MPI_COMM_RANK,
MPI_SEND,
MPI_RECV,
MPI_WTIME.
```

4.5.4 Measuring performances

The elapsed time (wall-clock) between two points in an MPI program can be measured by using operation `MPI_WTIME ()`. Its use is self-explanatory through a short segment of an MPI program example:

```
double start, finish;
start = MPI_Wtime ();
  ... //MPI program segment to be clocked
finish = MPI_Wtime ();
printf ("Elapsed time is %f\n", finish - start);
```

We are now ready to write a simple example of a useful MPI program that will measure the speed of communication channel between two processes. The program is presented, in more details, in the next subsection.

Example 4.3. Measuring communication bandwidth

Let us design a simple MPI program, which will measure the communication channel bandwidth, i.e. the amount of data transferred in a specified time interval, by using MPI communication operations `MPI_SEND` and `MPI_RECV`. As shown in Fig. 4.2, we will generate two processes, either on a single computer or on two interconnected computers. In the first case, the communication channel will be a data-bus that ”connects” the processes through their shared memory, while in the second case the communication channel will be an Ethernet link between computers.

The process with `rank = 0` will send a message, with a specified number of doubles, to the process with `rank = 1`. The communication time is a sum of the communication start-up time t_s and the message transfer time, i.e. the transfer time per word t_w times message length. We could expect that with shorter messages the bandwidth will be lower because a significant part of communication time will be spent on setting-up the software and hardware of the message communication channel, i.e. on the start-up time t_s. On the other hand, with long messages, the data

transfer time will dominate, hence, we could expect that the communication bandwidth will approach to a theoretical value of the communication channel. Therefore, the length of messages will vary from just a few data items to very long messages. The test will be repeated `nloop` times, with shorter messages, in order to get more reliable average results.

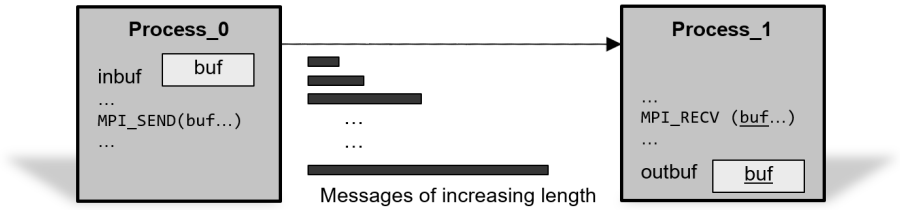


Fig. 4.2: A simple methodology for measurement of process-to-process communication bandwidth.

Considering the above methodology, an example of MPI program `MSMPIbw.cpp`, for measuring the communication bandwidth, is given in Listing 4.4. We have again a single program but slightly different codes for the sender and the receiver process. The essential part, message passing, starts in the sender process with a call to `MPI_Send`, which will be matched in the receiver process by a call to corresponding `MPI_Recv`.

```
#include "stdafx.h"
#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"
#define NUMBER_OF_TESTS 10  //for more reliable average results

int main(int argc, char* argv[])
{
  double       *buf;
  int          rank, numprocs;
  int          n;
  double       t1, t2;
  int          j, k, nloop;
  MPI_Status   status;
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
  if (numprocs != 2) {
    printf("The number of processes must be two!\n");
    return(0);
  }
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  if (rank == 0) {
    printf("\tn\ttime [sec]\tRate [Mb/sec]\n");
  }
  for (n = 1; n < 100000000; n *= 2) { //message length doubles
    nloop = 1000000 / n;
    if (nloop < 1) nloop = 1;  //just a single loop for long messages.
    buf = (double *)malloc(n * sizeof(double));
    if (!buf) {
      printf("Could not allocate message buffer of size %d\n", n);
```

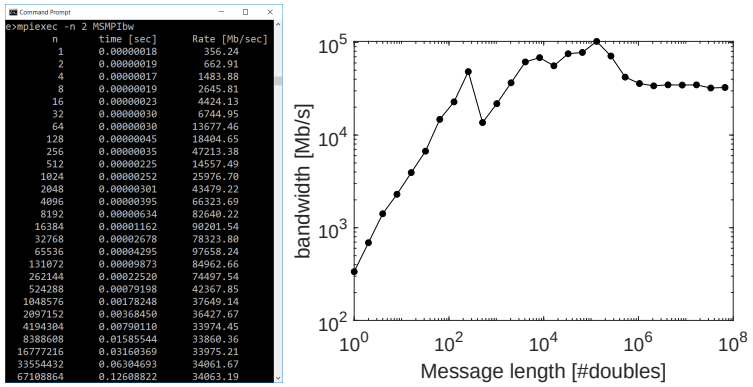
```
      MPI_Abort(MPI_COMM_WORLD, 1);
    }
    for (k = 0; k < NUMBER_OF_TESTS; k++) {
      if (rank == 0) {
        t1 = MPI_Wtime();
        for (j = 0; j < nloop; j++) {//send message nloop times
          MPI_Send(buf, n, MPI_DOUBLE, 1, k, MPI_COMM_WORLD);
        }
        t2 = (MPI_Wtime() - t1) / nloop;
      }
      else if (rank == 1) {
        for (j = 0; j < nloop; j++) {//receive message nloop times
          MPI_Recv(buf, n, MPI_DOUBLE, 0, k, MPI_COMM_WORLD, &status);
        }
      }
    }
    if (rank == 0) { //calculate bandwidth
      double bandwidth;
      bandwidth = n * sizeof(double)*1.0e-6 * 8 / t2; //in Mb/sec
      printf("\t%10d\t%10.8f\t%8.2f\n", n, t2, bandwidth);
    }
    free(buf);
  }
  MPI_Finalize();
  return 0;
}
```

Listing 4.4: MPI program for measuring bandwidth of a communication channel.

The output of the MPI program from Listing 4.4, which has been executed on two processes, each running on one of two computer cores that communicate through the shared memory, is shown in Fig. 4.3a with a screen-shot of `rank = 0` process user terminal, and in Fig. 4.3b with a corresponding bandwidth graph. The results confirmed our expectations. The bandwidth is poor with short messages and reaches the whole capacity of the memory access with longer messages.

If we assume that with very short messages, the majority of time is spent for the communication set-up, we can read from Fig. 4.3a (first line of data) that the set-up time was 0.35 μs. The set-up time starts increasing when the messages become longer than 32 of doubles. A reason could be that processes communicates until know through the fastest cache memory. Then the bandwidth increases until message length 512 of doubles. A reason for a drop at this length could be cache memory incoherences. The bandwidth converges then to 43 Gb/s, which could be a limit of cache memory access. If message lengths are increased above 524 thousands of doubles, the bandwidth is becoming lower and stabilizes at around 17 Gb/s, eventually because of a limit in shared memory access. Note, that the above merits are strongly related to a specific computer architecture and may therefore significantly differ among different computers. □

You are encourage to run the same program on your computer, and compare the obtained results with the results from Fig. 4.3. You may also run the same program on two interconnected computers, e.g. by Ethernet or Wi-Fi, and try to explain the obtained differences in results, taking into account a limited speed of your connection. Note, that the maximum message lengths `n` could be made shorter in the case of slower communication channels.



a) b)

Fig. 4.3: The bandwidth of a communication channel between two processes on a single computer that communicate through shared memory. a) message length, communication time and bandwidth, all in numbers; b) corresponding graph of the communication bandwidth.

4.6 Collective MPI communication

The communication operations, described in the previous sections, are called from a single process, identified by a `rank`, which has to be explicitly expressed in the MPI program, e.g. by a statement `if(my_id == rank)`. The MPI collective operations are called by all processes in a communicator. Typical tasks that can be elegantly implemented in this way are: global synchronization, reception of a local data item from all cooperating processes in the communicator and a lot of others, some of them described in this section.

4.6.1 `MPI_BARRIER (comm)`

This operation is used to synchronize the execution of a group of processes specified within the communicator `comm`. When a process reach this operation it has to wait until all other processes have reached the `MPI_BARRIER`. In other words, no process returns from `MPI_BARRIER` until all processes have called it. Note that the programmer is responsible that all processes from communicator `comm` will really call to `MPI_BARRIER`.

The barrier is a simple way of separating two phases of a computation to ensure that messages generated in different phases do not interfere. Note again, that the `MPI_BARRIER` is a global operation that invokes all processes therefore it could be

time-consuming. In many cases the call to MPI_BARRIER should be avoided by an appropriate use of explicit addressing options, e.g. tag, source, or comm.

4.6.2 MPI_BCAST (*inbuf*, incnt, intype, root, comm)

The operation implements a one-to-all broadcast operation whereby a single named process root sends its data to all other processes in the communicator, including to itself. Each process receives this data from the root process, which can be of any rank. At the time of call, the input data are located in inbuf of process root and consists of incnt data items of a specified intype. This implies that the number of data items must be exactly the same at input and output side. After the call, the data are replicated in inbuf as output data of all remaining processes. As *inbuf* is used as an input argument at the root process, but as an output argument in all remaining processes, it is of the *INOUT* type.

A schematic presentation of data broadcast after the call to MPI_BCAST is shown in Fig. 4.4 for a simple case of three processes, where the process with rank = 0 is the root process. Arrows symbolizes the required message transfer. Note, that all processes have to call MPI_BCAST to complete the requested data relocation.

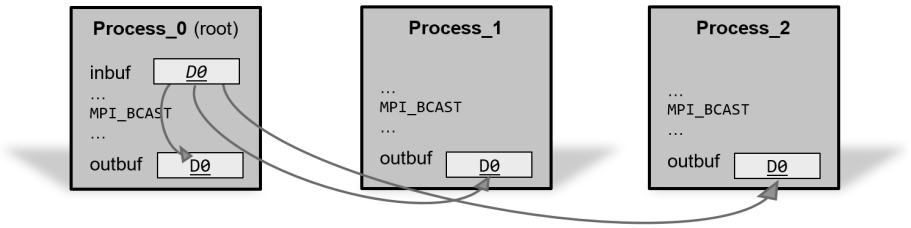


Fig. 4.4: Root process replicates the data from its input buffer in the output buffers of all processes.

Note, that the functionality of MPI_BCAST could be implemented, in the above example, by three calls to MPI_SEND in the root process and by a single corresponding MPI_RECV call in any process. Usually, such an implementation will be less efficient than the original MPI_BCAST. All collective communications could be time consuming. Their efficiency is strongly related with the topology and performance of interconnection network.

4.6.3 MPI_GATHER (inbuf, incnt, intype, outbuf, outcnt, outtype, root, comm)

All-to-one collective communication is implemented by MPI_GATHER. This operation is also called by all processes in the communicator. Each process, including root process, sends its input data located in inbuf that consists of incnt data items of a specified intype, to the root process, which can be of any rank. Note, that the communication data can be different in count and type for each process. However, the root process has to allocate enough space, through its output buffer, that suffices for all expected data. After the return from MPI_GATHER in all processes, the data are collected in outbuf of the root processes.

A schematic presentation of data relocation after the call to MPI_GATHER is shown in Fig. 4.5 for the case of three processes, where process with rank = 0 is the root process. Note again, that all processes have to call MPI_GATHER to complete the requested data relocation.

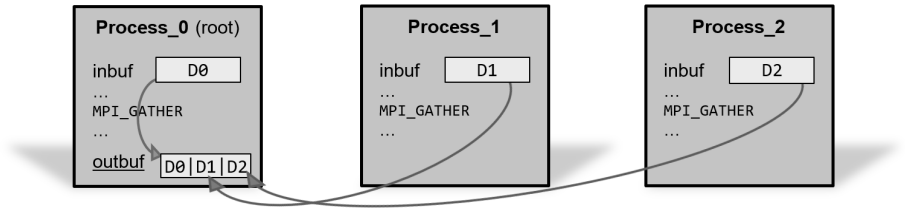


Fig. 4.5: Root process collects the data from input buffers of all processes in its output buffer.

4.6.4 MPI_SCATTER (inbuf, incnt, intype, outbuf, outcnt, outtype, root, comm)

This operation works inverse to MPI_GATHER, i.e. it scatters data from inbuf of process root to outbuf of all remaining processes, including itself. Note, that the count outcnt and type outtype of the data in each of the receiver processes are the same, so, data is scattered into equal segments.

A schematic presentation of data relocation after the call to MPI_SCATTER is shown in Fig. 4.6 for the case of three processes, where process with rank = 0 is the root process. Note again, that all processes have to call MPI_SCATTER to complete the requested data relocation.

There are also more complex collective operations, e.g. MPI_GATHERV and MPI_SCATTERV that allow a varying count of process data from each process and permit some options for process data placement on the root process. Such extensions

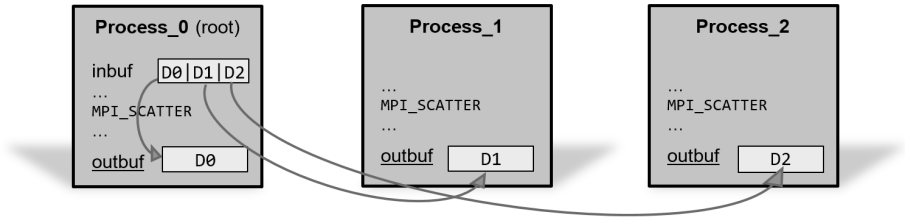


Fig. 4.6: Root process scatters the data from its input buffer to output buffers of all processes in its output buffer.

are possible by changing the `incnt` and `outcnt` arguments from a single integer to an array of integers, and by providing a new array argument `displs` for specifying the displacement relative to `root` buffers at which to place the processes' data.

4.6.5 Collective MPI data manipulations

Instead of just relocating data between processes, MPI provides a set of operations that perform several simple manipulations on the transferred data. These operations represent a combination of collective communication and computational manipulation in a single call and therefore simplify MPI programs.

Collective MPI operations for data manipulation are based on data reduction paradigm that involves reducing a set of numbers into a smaller set of numbers via a data manipulation. For example, three pairs of numbers: {5, 1}, {3, 2}, {7, 6}, each representing the local data of a process, can be reduced in a pair of maximum numbers, i.e. {7, 6}, or in a sum of all pair numbers, i.e. {15, 9}, and in the same way for other reduction operations defined by MPI:

- `MPI_MAX`, `MPI_MIN`; return either maximum or minimum data item,
- `MPI_SUM`, `MPI_PROD`; return either sum or product of aligned data items,
- `MPI_LAND`, `MPI_LOR`, `MPI_BAND`, `MPI_BOR`; return logical or bitwise AND or OR operation across the data items,
- `MPI_MAXLOC`, `MPI_MINLOC`; return the maximum or minimum value and the rank of the process that owns it.
- The MPI library enables to define custom reduction operations, which could be interesting for advanced readers (see references in Section 4.10 for details).

The MPI operation that implements all kind of data reductions is:

`MPI_REDUCE (inbuf, outbuf, count, type, op, root, comm).`

The `MPI_REDUCE` operation implements manipulation `op` on matching data items in input buffer `inbuf` from all processes in the communicator `comm`. The results of the manipulation are stored in the output buffer `outbuf` of process `root`. The

functionality of MPI_REDUCE is in fact an MPI_GATHER followed by manipulation op in process root. Reduce operations are implemented on a per-element basis, i.e. *i*-th elements from each process' inbuf are combined into the *i*-th element in outbuf of process root.

A schematic presentation of the MPI_REDUCE functionality before and after the call:

```
MPI_REDUCE (inbuf,outbuf,2,MPI_INT, MPI_SUM,0,MPI_COMM_WORLD)
```

is shown in Fig. 4.7. Before the call, inbuf of three processes with ranks 0, 1, and 2 were: {5, 1}, {3, 2}, and {7, 6}, respectively. After the call to the MPI_REDUCE the value in outbuf of root process is {15, 9}.

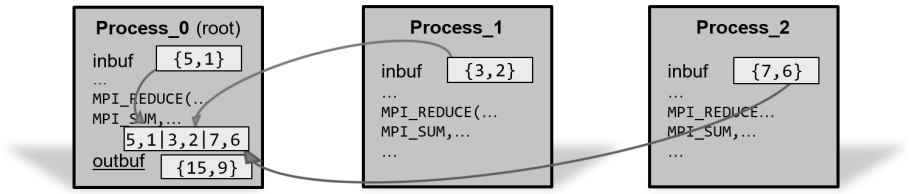


Fig. 4.7: Root process collects the data from input buffers of all processes, performs per-element MPI_SUM manipulation, and saves the result in its output buffer.

In many parallel calculations, a global problem domain is divided into subdomains that are assigned to corresponding processes. Often, an algorithm requires that all processes take a decision based on the global data. For example, an iterative calculation can stop when the maximal solution error reaches a specified value. An approach to the implementation of the stopping criteria could be the calculation of maximal subdomain errors and collection of them in a root process, which will evaluate a stopping criteria and broadcast the final result/decision to all processes. MPI provides a specialized operation for this task, i.e.:

```
MPI_ALLREDUCE (inbuf, outbuf, count, type, op, comm),
```

which improves simplicity and efficiency to MPI programs. It works as MPI_REDUCE followed by MPI_BCAST. Note that the argument root is not needed any more because the final result has to be available to all processes in the communicator. For the same inbuf data as in Fig. 4.7 and with MPI_SUM manipulation, a call to MPI_ALLREDUCE will produce the result {15, 9}, in output buffers of all processes in the communicator.

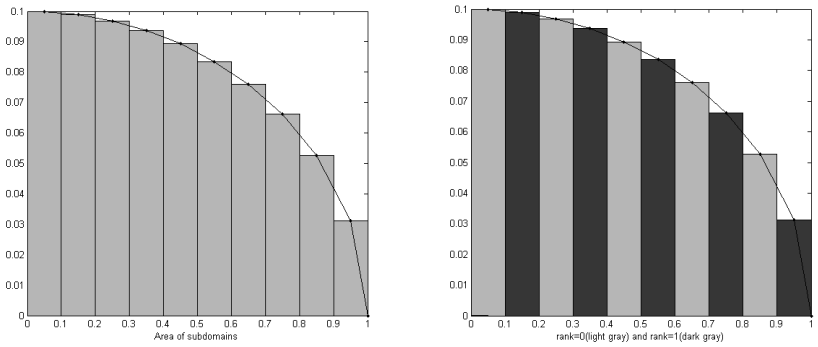
Example 4.4. Parallel computation of π

We know that for an efficient parallel execution on multiple processors implies that a complex task has to be decomposed in sub-tasks of similar complexity that have to be executed in parallel on all available processors. Consider again a computation of π by a numerical integration of $4\int_0^1\sqrt{1-x^2}dx$, which represents the area

of a circle with radius one that is equal to π. A detailed description of this task is given in Section 3. We divide the interval $[0,1]$ into N sub-intervals of equal width. The area of sub-intervals is calculated in this case slightly different, by a multiplication of sub-interval width with the evaluated integrand `yi` in the central point `xi` of each sub-interval. Finally, all sub-areas are summed-up by a partial sum. The schematic presentation of the described methodology is shown in Fig. 4.8 a) with ten sub-intervals with central points $xi = [0.05, 0.1, \ldots, 0.95]$.

If we have p available processes and p is much smaller than the number of sub-intervals N, which is usually the case if we need an accurate solution, the calculation load has to be distributed among all available processes, in a balanced way, for efficient execution. One possible approach is to assign each p-th sub-interval to a specific process. For example, a process with `rank = myID` will calculate the following sub-intervals: $i = myID + 1$, $i + p$, $i + 2p$, until $i + (k-1)p \leq N$, where $k = \lceil N/p \rceil$. In the case of $N \gg p$, and N is dividable by p, $k = N/p$, and each process calculates N/p intervals. Otherwise, a small unbalance in the calculation is introduced, because some processes have to calculate one additional sub-interval, while remaining processes will already finish their calculation. After the calculation of the area of sub-intervals, p partial sums are reduced to a global sum, i.e. by sending MPI messages to a root process. The global sum approximates π, which should be now computed faster and more accurate if more intervals are used.

A simple case for two processes and ten intervals is shown in Fig. 4.8 b). Five sub-intervals {1,3,5,7,9}, marked in grey, are integrated by `rank` 0 process and the other five sub-intervals {2,4,6,8,10}, marked in dark, are integrated by `rank` 1 process.



a) b)

Fig. 4.8: a) Discretization of interval $[0,1]$ in 10 sub-intervals for numerical integration of quarter circle area; b)Decomposition for two parallel processes: light-grey sub-intervals are sub-domain of `rank` 0 process; dark-grey sub-intervals of `rank` 1 process.

An example of an MPI program that implements parallel computation of π, for an arbitrary p and N, in C programming language, is given in Listing 4.5:

```
#include "stdafx.h"
#include <stdio.h>
#include <math.h>
#include "mpi.h"
int main(int argc, char *argv[])
{
  int done = 0, n, myid, numprocs, i;
  double PI25DT = 3.14159265358979323846264 3;
  double pi, h, sum, x, start, finish;
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD, &myid);
  while (!done) {
    if (myid == 0) {
      printf("Enter the number of intervals: (0 quits) ");
      fflush(stdout);
      scanf_s("%d", &n);
      start = MPI_Wtime();
    }
    //execute in all active processes
    MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
    if (n == 0) done = 1;
    h = 1.0 / (double)n;
    sum = 0.0;
    for (i = myid + 1; i <= n; i += numprocs) {
      x = h * ((double)i - 0.5);
      sum += 4.0 * h * sqrt(1.0 - x*x);
    }
    MPI_Reduce(&sum, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
    if (myid == 0) {
      finish = MPI_Wtime();
      printf("Pi is app. %.16f, Error is %.16f\n",pi,fabs(pi-PI25DT));
      printf("Elapsed time is %f\n", finish - start);
    }
  }
  MPI_Finalize();
  return 0;
}
```

Listing 4.5: MPI program in C for parallel computation of π.

Let us open a Terminal window and a Task manager window (see Fig. 4.9), where we see that the computer used has four cores, eight logical processors and is utilized by a background task for 17%. After running the compiled program for the calculation of π value on a single process and with 10^9 intervals, the execution time is about 31.4s and the CPU utilization increases to 30%. In the case of four processes, the execution time drops to 7.9s and utilization increases to 70%. Running the program on 8 processes further speed-up is noticed, by the execution time 5.1s and CPU utilization 100%, because all computational resources of the computer are now fully utilized. From prints in the terminal window, it is evident that the value π was calculated with similar accuracy in all cases. With our simple MPI program, we achieved a speed-up a bit higher than 6, which is excellent!

□

Recall, that we have parallelized the computation of π by distributing the computation of sub-intervals areas among cooperating processes. In this simple example,

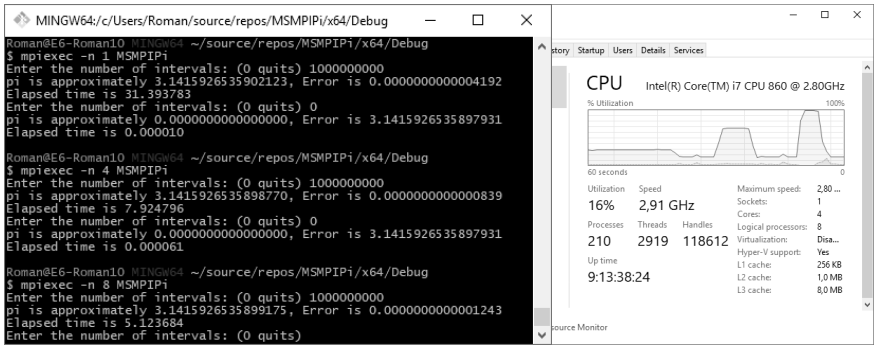


Fig. 4.9: Screenshots of Terminal window and Task manager indicating timing of the program for calculation of π and the computer utilization history.

our, initially continuous, computation domain was interval $[0,1]$. The domain was disretized into N sub-intervals. Then a 1-D domain decomposition was used to divide the whole domain into p sub-domains, where p is the number of cooperating processes that did the actual computation. Finally, the partial results have been assembles in a selected host process and output as a final result. This is the most often used approach for the paralelization in numerical analysis. It can be applied for the operations on large vectors or matrices, for solutions of systems of equations, for solutions of partial differential equations (PDE), and similar. A more detailed methodology and analysis pf the parallel program will be given in Part III.

4.7 Communication and computation overlap

Contemporary computers have separate communication and calculation resources, therefore they are able to execute both tasks in parallel, which is a significant potential for improving an MPI program efficiency. For example, instead of just waiting for a data transmission to be completed, a certain part of calculation could be done that could be eventually required in the next computing step. If a process can perform useful work while some long communication is in progress, overall execution time might be reduced. This approach is often termed as a hiding latency.

Various communication modes are available in MPI that enable hiding latency, but they require correct usage to avoid communication deadlock or program shutdown. The measures for managing potential deadlocks of communication operations are addressed in more details in a separate subsection. Finally, a single MPI program in the ecosystems of more communicators is presented. More advanced topics, e.g. a virtual shared memory emulation through so called MPI windows, which could simplify the programming and improve the execution efficiency, are beyond the

Quite enough MPI operations

We are now quite familiar with enough operations for coding simpler MPI programs and for evaluating their performances. A list of corresponding MPI operations is shown below:

Basic MPI operations:

```
MPI_INIT, MPI_FINALIZE,
MPI_COMM_SIZE, MPI_COMM_RANK,
MPI_SEND, MPI_RECV,
```

MPI operations for collective communication:

```
MPI_BARRIER,
MPI_BCAST, MPI_GATHER, MPI_SCATTER,
MPI_REDUCE, MPI_ALLREDUCE,
```

Control MPI operations:

```
MPI_WTIME, MPI_STATUS,
MPI_INITIALIZED.
```

scope of this book and are well covered by continual evolving MPI standard, which should be ultimate reference of enthusiastic programmers.

4.7.1 Communication modes

MPI processes can communicate in four different communication modes: standard, buffered, synchronous and ready. Each of these modes can be performed in blocking or in non-blocking type, first being less eager to the amount of required memory for message buffering, and second being often more efficient, because of an ability to overlap the communication and computation tasks.

Blocking communication

A **standard mode** send call, described in Section 4.5 with operation `MPI_SEND`, should be assumed as a blocking send, which will not return until the message data and envelope have been safely stored away. The sender process can access and overwrite the send buffer with a new message. However, depending on the MPI implementation, short messages might still be buffered while longer messages might be

split and sent in shorter fragments, or they might be copied into a temporary communication buffer (see Fig.4.1 for details).

Because the message buffering requires extra memory space and memory-to-memory copying, implementations of MPI libraries do not guarantee the amount of buffering therefore one has always to count on the possibility that send call will not complete until a matching receive has been posted, and the data has been moved to the receiver. In other words, the standard send call is `non-local`, i.e. may require execution of an MPI operation in another process.

According to the MPI standard, a program is correct and portable if it does not rely on system buffering in the standard mode. Buffering may improve the performance of a correct program, but does not affect the result of the program. There are three blocking send call modes, indicated by a single-letter prefix: `MPI_BSEND`, `MPI_SSEND`, `MPI_RSEND`, with `B` for buffered, `S` for synchronous, and `R` for ready, respectively. The send operation syntax is the same as in the standard send, e.g. `MPI_BSEND (buf, count, datatype, dest, tag, comm)`.

The **buffered mode** send is a standard send with a user-supplied message buffering. It will start independent of a matching receive and can complete before a matching receive is posted. However, unlike the standard send, this operation is **local**, i.e. its completion is independent on the matching receive. Thus, if a buffered send is executed and no matching receive is posted, then the MPI will buffer the outgoing message, to allow the send call to complete. It is a responsibility of programmer to allocate enough buffer space for all subsequent `MPI_BSEND` by calling `MPI_BUFFER_ATTACH (bbuf, bsize)`. The buffer space `bbuf` cannot be reused by subsequent `MPI_BSEND`s if they have not been completed by matching `MPI_RECV`s, therefore it must be large enough to store all subsequent messages.

The **synchronous** mode send can start independently of a matching receive. However, the send will complete successfully only if a matching receive operation has started to receive the message sent by the synchronous send. Thus, the completion of a synchronous send not only indicates that the send buffer can be reused, but also indicates that the receiver has reached a certain point in its execution, i.e. it has started executing the matching receive. If both sends and receives are blocking operations then the use of the synchronous mode provides synchronous communication semantics: a communication does not complete at either end before both processes rendezvous at the communication. A send executed in this mode is **non-local**, because its competition requires a cooperation of sender and receiver processes.

The **ready mode** send may be started only if the matching receive has been already called. Otherwise, the operation is erroneous and its outcome is undefined. On some systems, this allows the removal of a hand-shake operation that is otherwise required, which could result in improved performance. In a correct program, a ready send can be replaced by a standard send with no effect on the program results, but with eventual improved performances.

The receive call `MPI_RECV` is always blocking, because it returns only after the receive buffer contains the expected received message.

Non-blocking communication

Non-blocking send start calls are denoted by a leading letter `I` in the name of MPI operation. They can use the same four modes as blocking sends: standard, buffered, synchronous and ready, i.e. `MPI_ISEND`, `MPI_IBSEND`, `MPI_ISSEND`, `MPI_IRSEND`. Sends of all modes, except ready, can be started whether a matching receive has been posted or not; a non-blocking ready send can be started only if a matching receive is posted. In all cases, the non-blocking send start call is **local**, i.e. it returns immediately, irrespective of the status of other processes. Non-blocking communications return immediately request handles that can be waited on, or queried, by specialized MPI operations that enables to wait or to test for their completion.

The syntax of the non-blocking MPI operations are the same as in the standard communication mode, e.g.:
`MPI_ISEND (buf, count, datatype, dest, tag, comm, request)`, or
`MPI_IRECV (buf, count, datatype, dest, tag, comm, request)`,
except with an additional request handle that is used for later querying by send-complete calls, e.g.:
`MPI_WAIT (request, status)`, or
`MPI_TEST (request, flag, status)`

A non-blocking **standard send** call `MPI_ISEND` initiates the send operation, but does not complete it, in a sense that it will return before the message is copied out of the send buffer. A later separate call is needed to complete the communication, i.e. to verify that the data has been copied out of the send buffer. In the meantime, a computation can run concurrently. In the same way, a non-blocking receive call `MPI_IRECV` initiates the receive operation, but does not complete it. The call will return before a message is stored into the receive buffer. A later separate call is needed to verify that the data has been received into the receive buffer. While querying about the reception of the complete message, a computation can run concurrently.

We can expect that a non-blocking send `MPI_ISEND` immediately followed by send-complete call `MPI_WAIT` is functionally equivalent to a blocking send `MPI_SEND`. One can wait on multiple requests, e.g. in a master/slave MPI program, where the master waits either for all or for some slaves' messages, using MPI operations:
`MPI_WAITALL (count, array_of_requests, array_of_statuses)`, or
`MPI_WAITSOME (incount, array_of_requests, outcount, array_of_indices, array_of_statuses)`.

A send-complete call returns when data has been copied out of the send buffer. It may carry additional meaning, depending on the send mode. For example, if the send mode is synchronous, then the send can complete only if a matching receive has started, i.e. a receive has been posted, and has been matched with the send. In this case, the send-complete call is non-local. Note that a synchronous, non-blocking send may complete, if matched by a non-blocking receive, before the receive complete call occurs. It can complete as soon as the sender "knows" that the transfer will complete, but before the receiver "knows" that the transfer will complete.

If the non-blocking send is in *buffered mode* then the message must be buffered if there is no pending receive. In this case, the send-complete call is local, and must succeed irrespective of the status of a matching receive. If the send mode is standard then the send-complete call may return before a matching receive occurred, if the message is buffered. On the other hand, the send-complete may not complete until a matching receive occurred, and the message was copied into the receive buffer. Some further facts or implications of the non-blocking communication mode are listed below. Non-blocking sends can be matched with blocking receives, and vice-versa. The completion of a send operation may be delayed, for a standard mode, and must be delayed, for synchronous mode, until a matching receive is posted. The use of non-blocking sends in these two cases allows the sender to proceed ahead of the receiver, so that the computation is more tolerant of fluctuations in the speeds of the two processes.

Non-blocking sends in the buffered and ready modes have a more limited impact. A non-blocking send will return as soon as possible, whereas a blocking send will return after the data has been copied out of the sender memory. The use of non-blocking sends is advantageous in these cases only if data copying can be concurrent with computation.

The message-passing model implies that a communication is initiated by the sender. The communication will generally have lower overhead if a receive is already posted when the sender initiates the communication, e.g. message data can be moved directly into the receive buffer, and there is no need to queue a pending send request. However, a receive operation can complete only after the matching send has occurred. The use of non-blocking receives allows one to achieve lower communication overheads without blocking the receiver while it waits for the send. There are further, more advanced, approaches for optimized use of the communication modes that are beyond the scope of this chapter; however, they are well documented elsewhere (see Section 4.10).

4.7.2 Sources of deadlocks

We know from previous sections that after a call to receive operation, e.g. `MPI_RECV`, the process will wait patiently until a matching `MPI_SEND` is posted. If the matching send is never posted, the receive operation will wait forever in a deadlock. In practice, the program will become unresponsive until some time limit is exceeded, or the operating system will crash. The above situation can appear if two `MPI_RECV` are issued in approximately the same time, on two different processes, that mutually expect a matching send and are waiting to the matching messages, that will never start and therefore never deliver the expected messages. Such a situation is shown below with a segment from an MPI program, in C language, for process with `rank = 0` and `rank =1`, respectively:

```
if (rank == 0) {
  MPI_Recv (rec_buf, count, MPI_BYTE, 1, tag, comm, &status);
```

```
    MPI_Send (send_buf, count, MPI_BYTE, 1, tag, comm);
  }
  if (rank == 1) {
    MPI_Recv (rec_buf, count, MPI_BYTE, 0, tag, comm, &status);
    MPI_Send (send_buf, count, MPI_BYTE, 0, tag, comm);
  }
```

In the same way, if two blocking MPI_SENDs are issued in approximately the same time, on process, e.g. with rank = 0 and rank =1, respectively, both followed by a matching MPI_RECV, they will never finish if MPI_SENDs are implemented without buffers. Even in the case that message buffering is implemented, it will usually suffices only for shorter messages. With longer messages, a deadlock situation could be expected, when the buffer space is exhausted, which was already demonstrated in Listing 4.3.

The above situations are called ”unsafe” because they depend on the implementation of the MPI communication operations and on the availability of system buffers. The portability of such unsafe programs may be limited.

Several solutions that can make an unsafe program ”correct”. The simplest approach is to use the order of communication operations more carefully. For example, in the given example, by a call to MPI_RECV, in process with rank = 0, first. Consequently, with exchanging the order of two lines in the program segment for process with rank = 0:

```
  if (rank == 0) {
    MPI_Send (send_buf, count, MPI_BYTE, 1, tag, comm);
    MPI_Recv (rec_buf, count, MPI_BYTE, 1, tag, comm, &status);
  }
  if (rank == 1) {
    MPI_Recv (rec_buf, count, MPI_BYTE, 0, tag, comm, &status);
    MPI_Send (send_buf, count, MPI_BYTE, 0, tag, comm);
  }
```

send and receive operations are automatically matched and deadlocks are avoided in both processes.

An alternative approach is to supply receive buffer in the same time as the send buffer, which can be done by operation MPI_SENDRECV. If we replace the MPI_RECV and MPI_SEND pair by MPI_SENDRECV, in both processes, the deadlock is not possible, because four buffers will prevent eventual mutual waiting.

Next possibility is to use a pair of non-blocking operations MPI_IRECV, MPI_ISEND in each process, with subsequent waiting in both processes to both requests by MPI_WAITALL:

```
  ...
  MPI_Request requests[2]
  ...
  if (rank == 0) {
    MPI_Irecv (rec_buf,count,MPI_BYTE,1,tag,comm,&requests[0]);
```

```
    MPI_Isend(send_buf,count,MPI_BYTE,1,tag,comm,&requests[1]);
  }
  else if (rank == 1) {
    MPI_Irecv (rec_buf,count,MPI_BYTE,0,tag,comm,&requests[0]);
    MPI_Isend(send_buf,count,MPI_BYTE,0,tag,comm,&requests[1]);
  }
  MPI_Waitall (2, request, MPI_STATUSES_IGNORE);
```

The call to MPI_IRECV is issued first, which provides a receive data buffer that is ready for the message that will arrive. This approach avoids extra memory copies of data buffers, avoids deadlock situations and could therefore speed-up the program execution.

Finally, non-blocking buffered send can be used MPI_BSEND with explicit allocation of separate send buffers by MPI_BUFFER_ATTACH, however, this approach needs extra memory.

Example 4.5. Hiding latency

We have learned that a blocking send will continue to wait until a matching receive will signal that it is ready to receive. In situations where a significant calculation work follows a send of a large message, and it does not interfere with the send buffer, it might be more efficient to use non-blocking send. Now, the calculation work following the send operation can start almost immediately after the send process is initiated, and can continue to run while the send operation is pending. Similarly, a non-blocking receive could be more efficient than its blocking counter-part if work following the receive operation does not depend on the received message.

In some MPI programs, communication and calculation tasks can run concurrently, and consequently, can speed-up the program execution. Suppose that a master process have to receive large messages from all slaves. Then, all processes have to do an extensive calculation that is independent of data in the messages. If blocking communication is used, the execution time will be a sum of communication and calculation time. If asynchronous, non-blocking communication is used, a part of communication and calculation tasks could overlap, which could result in a shorter execution time.

One way to implement the above task is to start a master process that will receiving messages from all slave processes, and then proceed with its calculation work. The slave processes will send their messages and then start to calculate. The program runs until all communication and calculation is done. A simple demonstration code of overlapping communication and calculation is given in Listing 4.6.

```
1 #include <mpi.h>
2 #include <stdlib.h>
3 #include <math.h>
4 #include <stdio.h>
5
6 double other_work(int numproc)
7 {
8   int i;  double a;
9   for (i = 0; i < 100000000/numproc; i++) {
```

```
    a = sin(sqrt(i));   //different amount of calculation
  }
  return a;
}

int main(int argc, char* argv[])   //number of processes must be > 1
{
  int p, i, myid, tag=1, proc, ierr;
  double start_p, run_time, start_c, comm_t, start_w, work_t, work_r;
  double *buff = nullptr;

  MPI_Request request;
  MPI_Status status;

  MPI_Init(&argc, &argv);
  start_p = MPI_Wtime();
  MPI_Comm_rank(MPI_COMM_WORLD, &myid);
  MPI_Comm_size(MPI_COMM_WORLD, &p);

  #define master    0
  #define  MSGSIZE  100000000 //5000000  //different sizes of messages
  buff = (double*)malloc(MSGSIZE * sizeof(double));   //allocate

  if (myid == master) {
    for (i = 0; i < MSGSIZE; i++) { //initialize message
      buff[i] = 1;
    }
    start_c = MPI_Wtime();
    for (proc = 1; proc<p; proc++) {
#if 1
      MPI_Irecv(buff, MSGSIZE,     //non-blocking receive
        MPI_DOUBLE, MPI_ANY_SOURCE,  tag, MPI_COMM_WORLD, &request);
#endif
#if 0
      MPI_Recv(buff, MSGSIZE,     //blocking receive
         MPI_DOUBLE, MPI_ANY_SOURCE, tag, MPI_COMM_WORLD, &status);
#endif
    }
    comm_t = MPI_Wtime() - start_c;
    start_w = MPI_Wtime();
    work_r = other_work(p);
    work_t = MPI_Wtime() - start_w;
    MPI_Wait(&request, &status);    //block until Irecv is done
  }
  else {   //slave processes
    start_c = MPI_Wtime();
#if 1
    MPI_Isend(buff, MSGSIZE,     //non-blocking send
       MPI_DOUBLE, master, tag, MPI_COMM_WORLD, &request);
#endif
#if 0
    MPI_Send(buff, MSGSIZE,     //blocking send
       MPI_DOUBLE, master, tag, MPI_COMM_WORLD);
#endif
    comm_t = MPI_Wtime()-start_c;
    start_w = MPI_Wtime();
    work_r = other_work(p);
    work_t = MPI_Wtime()-start_w;
    MPI_Wait(&request, &status);    //block until Isend is done
  }
  run_time = MPI_Wtime() - start_p;
  printf("Rank \t Comm[s] \t Calc[s] \t Total[s] \t Work_result\n");
  printf(" %d\t %e\t %e\t %e\t %e\t\n", myid, comm_t, work_t, run_time,↩
     work_r);
  fflush(stdout);  //to correctly finish all prints
  free(buff);
  MPI_Finalize();
```

```
76 }
```

Listing 4.6: Communication and calculation overlap

The program from Listing 4.6 has to be executed with at least two processes, one master and one or more slaves. The non-blocking `MPI_Isend` call, in all processes, returns immediately to the next program statement without waiting for the communication task to complete. This enables tt other_work to proceed without delay. Such a usage of non-blocking send (or receive), to avoid processor idling, has the effect of "latency hiding", where MPI latency is the elapse time for an operation, e.g. `MPI_Isend`, to complete. Note, that we have used `MPI_ANY_SOURCE` in the master process to specify message source. This enables an arbitrary arrival order of messages, instead of a predefined sequence of processes, that can further speed-up the program execution.

The output of this program should be as follows:

```
$ mpiexec -n 2 MPIhiding
Rank  Comm[s]       Calc[s]       Total[s]      Work_result
 1    2.210910e-04  1.776894e+00  2.340671e+00  6.109991e-01
Rank  Comm[s]       Calc[s]       Total[s]      Work_result
 0    1.692562e-05  1.747064e+00  2.340667e+00  6.109991e-01
```

Note, that the total execution time is longer than the calculation time. Te communication time is negligable, even that we have sent 100 millions of doubles. Please use blocking MPI communication, compare the execution time and explain differences. Please experiment with different number of processes, different message length, different amount of calculation, and explain the behavior of the execution time.

□

4.7.3 Some subsidiary features of message-passing

The MPI communication model is by default **non-deterministic**. The arrival order of messages sent from two processes, A and B, to a third process, C, is not known in advance.

The MPI communication is **unfair**. No matter how long a send process has been pending, it can always be overtaken by a message sent from another sender process. For example, if process A sends a message to process C, which executes a matching receive operation, and process B sends a competing message that also matches the receive operation in process C, only one of the sends will complete. It is the programmer's responsibility to prevent "starvation" by ensuring that a computation is deterministic, e.g. by forcing a reception of specific number of messages from all competing processes.

The MPI communication is **non-overtaking**. If a sender process posts successive messages to a receiver process, and a receive operation matches all messages, the

messages will be managed in the order as they were sent, i.e. the first sent message will be received first, etc. Similarly, if a receiver process posts successive receives, and all match the same message, then the messages will be received in the same order as they have been sent. This requirement facilitates correct matching of a send to a receive operation and guarantees that an MPI program is deterministic, if the cooperating processes are single-threaded.

On the other hand, if an MPI process is multi-threaded, then the semantics of thread execution may not define a relative order between send operations from distinct program threads. In the case of multi-threaded MPI processes, the messages sent from different threads can be received in an arbitrary order. The same is valid also for multi-threaded receive operations, i.e. successively sent messages will be received in an arbitrary order.

Example 4.6. Fairness and overtaking of MPI communication

A simple demonstration example of some MPI communication features is given in Listing 4.7. The master process is ready to receive `10*(size-1)` messages, while each of the slave processes want to send 10 messages to the master process, each with a larger tag. The master process lists all received messages with their source process ranks and their tags.

```
#include "stdafx.h"
#include "mpi.h"
#include <stdio.h>
int main(int argc, char **argv)
{
  int rank, size, i, buf[1];
  MPI_Status status;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  if (rank == 0) {
    for (i = 0; i < 10*(size-1); i++) {
        MPI_Recv(buf, 1, MPI_INT, MPI_ANY_SOURCE,
        MPI_ANY_TAG, MPI_COMM_WORLD, &status);
      printf("Msg from %d with tag %d\n",
        status.MPI_SOURCE, status.MPI_TAG);
    }
  }
  else { //rank > 0
    for (i = 0; i < 10; i++)
      MPI_Send(buf, 1, MPI_INT, 0, i, MPI_COMM_WORLD);
  }
  MPI_Finalize();
  return 0;
}
```

Listing 4.7: Demonstration of unfairness and non-overtaking in MPI communication.

The output of this program depends on the number of cooperating processes. For the case of 3 processes it should be as follows:

```
$ mpiexec -n 3 MPIfairness
Msg from 1 with tag 0
```

```
Msg from 1 with tag 1
Msg from 1 with tag 2
Msg from 1 with tag 3
Msg from 1 with tag 4
Msg from 1 with tag 5
Msg from 1 with tag 6
Msg from 1 with tag 7
Msg from 1 with tag 8
Msg from 1 with tag 9
Msg from 2 with tag 0
Msg from 2 with tag 1
Msg from 2 with tag 2
Msg from 2 with tag 3
Msg from 2 with tag 4
Msg from 2 with tag 5
Msg from 2 with tag 6
Msg from 2 with tag 7
Msg from 2 with tag 8
Msg from 2 with tag 9
```

We see that all messages from process with rank 1 have been received first, even that the process with rank 2 has also attempted to send its messages, so the communication was unfair. The order of received messages, identified by tags, is the same as the order of sent messages, so the communication was non-overtaking. □

4.7.4 MPI communicators

All communication operations introduced in previous sections have used the default communicator `MPI_COMM_WORLD`, which incorporates all processes involved and defines a default context. More complex parallel programs usually need more process groups and contexts to implement various forms of sequential or parallel decomposition of a program. Also, cooperation of different software developer groups is much easier if they develop their software modules in distinct contexts. The MPI library supports modular programming via its communicator mechanism that provides the ”information hiding” and ”local name space”, which are both needed in modular programs.

We know from previous sections that any MPI communication operation specifies a **communicator**, which identifies a **process group** that can be engaged in the communication and a **context** (tagging space) in which the communication occurs. Different communicators can encapsulate the same or different process groups but always with different context. The message context can be implemented as an extended tag field, which enable to distinguish between messages from different context. A communication operation can receive a message only if it was sent in

the same context, therefore, MPI processes that run in different contexts can not be interfered by unwanted messages.

For example, in master-slave parallelization, master process manages the tasks for slave processes. To distinguish between master and slave tasks, statements like `if(rank==master)` and `if(rank>master)` for ranks in a default communicator `MPI_COMM_WORLD` can be used. Alternatively, the processes of a default communicator can be splitted in two new sub-communicators, each with a different group of processes. First group of processes, eventually with a single process, performs master tasks and the second group of processes, eventually with larger number of processes, executes slave tasks. Note, that both sub-communicators are encapsulated into a new communicator, while the default communicator still exists. A collective communication is possible now in the default communicator or in the new communicator.

In a further example, a sequentially decomposed parallel program is schematically shown in Fig. 4.10. Each of the three vertical lines with blocks represents a single process of the parallel program, i.e. `P_0`, `P_1`, and `P_2`. All three processes form a single process group. The processes are decomposed in consecutive sequential program modules shown with blocks. Process-to-process communication calls are shown with arrows. In Fig. 4.10a, all processes and their program modules run in the same context, while in Fig. 4.10b, program modules, encircled by dashed curves, run in two different contexts that were obtained by a duplication of the default communicator.

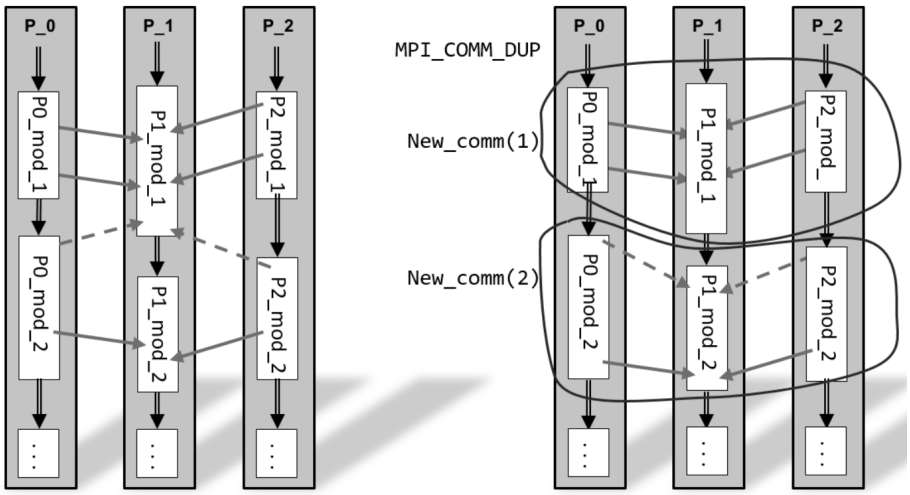


Fig. 4.10: Sequentially decomposed parallel program that runs on three processes. a) processes run in the same context; b) processes run in two different contexts.

Fig. 4.10a shows that MPI processes P_0 and P_2 have finished sooner than P_1. Dashed arrows denote messages that have been generated during subsequent computation in P_0 and P_2. The messages could be accepted by a sequential program module P1_mod_1 of MPI process P_1, which is eventually NOT correct. A problem solution is shown in Fig. 4.10b. The program modules run here in two different contexts, New_comm(1) and New_comm(2). The early messages will be accepted now correctly by MPI receive operations in program module P1_mod_2 from communicator New_comm(2), which uses a distinct tag space that will correctly match the problematic messages.

The MPI standard specifies several operations that support modular programming. Two basic operations implement duplication or splitting of an existing communicator comm.

MPI_COMM_DUP (comm, new_comm)

is executed by each process from the parent communicator comm. It creates a new communicator new_comm comprising the same process group but a new context. This mechanism supports sequential composition of MPI programs, shown in Fig. 4.10, by separating communication that is performed for different purposes. Since all MPI communication is performed within a specified communicator, MPI_COMM_DUP provides an effective way to create a new user-specified communicator, e.g. for use by a specific program module or by a library, in order to prevent interferences of messages.

MPI_COMM_SPLIT (comm, color, key, new_comm)

creates a new communicator new_comm from the initial communicator comm, comprising disjoint subgroups of processes with optional reordering of their ranks. Each subgroup contains all processes of the same color, which is a non-negative argument. It can be MPI_UNDEFINED; in this case its corresponding process will not be included in any of the new communicators. Within each subgroup, the processes are ranked in the order defined by the value of corresponding argument key, i.e. a lower value of key implies a lower value of rank, while equal process keys preserve the original order of ranks. A new sub-communicator is created for each subgroup and returned as a component of a new communicator new_comm. This mechanism supports parallel decomposition of MPI programs. The MPI_COMM_SPLIT is a collective communication operation with a functionality similar to MPI_ALLGATHER to collect color and key from each process. Consequently, the MPI_COMM_SPLIT operation must be executed by every process from the parent communicator comm, however, every process is permitted to apply different values for color and key.

Remember that every call to MPI_COMM_DUP or MPI_COMM_SPLIT should be followed by a call to MPI_COMM_FREE, which deallocates a communicator that can be reused later. The MPI library can create a limited number of objects at a time and not freeing them could result in a runtime error.

More flexible ways to create communicators, are based on MPI object MPI_GROUP. A process group is an ordered set of process identifiers associated with an integer

rank. Process groups allow a subset of processes to communicate among themselves using local names and identifiers without interfering with other processes, because groups do not have a context. Dedicated MPI operations can create groups in a communicator by MPI_COMM_GROUP, obtain a group size of a calling process by MPI_GROUP_SIZE, perform set operations between groups, e.g. union by MPI_GROUP_UNION, etc., and create a new communicator from the existing group by MPI_COMM_CREATE_GROUP.

There are many advanced MPI operations that support creation of communicators and structured communication between processes within a communicator, i.e. intra-communicator communication, and between processes from two different communicators, i.e. inter-communicator communication. These topics are useful for advanced programming, e.g. in the development of parallel libraries, which are not covered in this book.

Example 4.7. Splitting MPI communicators

Let visualize now the presented concepts with a simple example. Suppose that we would like to split a default communicator with eight processes ranked as rank = {0 1 2 3 4 5 6 7} to create two sets of process by a call to MPI_COMM_SPLIT, as shown in Fig. 4.11. Two disjoint sets should include processes with odd and even ranks, respectively. We therefore need two colors that can be created, for example, with division of original ranks by modulo 2: color = rank%2, which results in corresponding processes' colors = {0 1 0 1 0 1 0 1}. Ranks of processes in new groups are assigned according to process key. If corresponding keys are {0 0 0 0 0 0 0 0}, new process ranks in groups, new_g1 and new_g2, are sorted in the ascending order as in the initial communicator.

A call to MPI_COMM_SPLIT (MPI_COMM_WORLD, rank%2, 0, &new_comm); will partition the initial communicator with eight processes in two groups with four processes, based on the color, which is, in this example, either 0 or 1. The groups, identified by their initial ranks, are new_g1 = {0 2 4 6} and new_g2 = {1 3 5 7}. Because all process keys are 0, new process ranks of both groups are sorted in ascending order as rank = {0 1 2 3}.

A simple MPI program MSMPIsplitt.cpp in Listing 4.8 implements the above ideas. MPI_COMM_SPLIT is called by color = rank%2, which is either 0 or 1. Consequently, we get two process groups with four processes per group. Note, that the new process ranks in both groups are equal to {0 1 2 3}, because the key = 0 in all processes, and consequently, the original order of ranks remain the same. For an additional test, the master process calculates the sum of processes' ranks in each new group of a new communicator, using the MPI_REDUCE operation. In this simple example, the sum of ranks in both groups should be equal to $0+1+2+3=6$.

```
#include "stdafx.h"
#include <stdio.h>
#include "mpi.h"

int main(int argc, char **argv)
{
  int  numprocs, org_rank, new_size, new_rank;
```

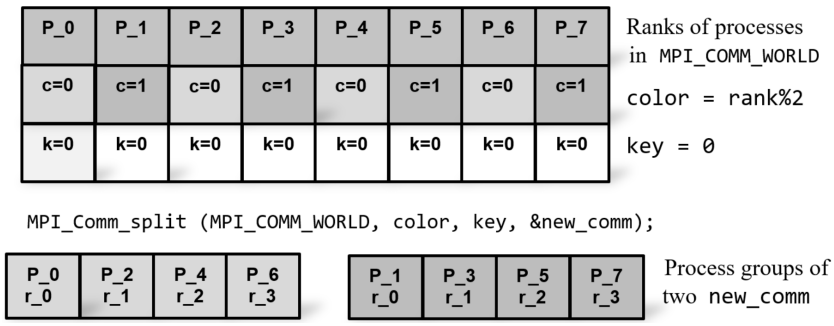


Fig. 4.11: Visualization of splitting the default communicator with eight processes in two sub-communicators with disjoint sets of four processes.

```
8   MPI_Comm new_comm;
9
10  MPI_Init(&argc, &argv);
11  MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
12  MPI_Comm_rank(MPI_COMM_WORLD, &org_rank);
13
14  MPI_Comm_split(MPI_COMM_WORLD, org_rank%2, 0, &new_comm);
15 //  MPI_Comm_split(MPI_COMM_WORLD,org_rank>=2,org_rank<=3,&new_comm);
16  MPI_Comm_size(new_comm, &new_size);
17  MPI_Comm_rank(new_comm, &new_rank);
18  printf("'MPI_COMM_WORLD' process rank/size %d/%d has rank/size %d/%d ↩
       in 'new_comm'\n", org_rank, numprocs, new_rank, new_size);
19
20  int sum_ranks; //calculate sum of ranks in both new groups of new_com
21  MPI_Reduce(&new_rank, &sum_ranks, 1, MPI_INT, MPI_SUM, 0, new_comm);
22  if (new_rank == 0) {
23    printf("Sum of ranks in 'new_com': %d\n", sum_ranks);
24  }
25
26  MPI_Comm_free(&new_comm);
27  MPI_Finalize();
28  return 0;
29 }
```

Listing 4.8: Splitting a default communicator in two process groups of a new communicator. First group and second process groups include, respectively, processes with even and odd ranks from the default communicator.

The output of compiled program from Listing 4.8, after running it on eight processes, should be similar to:

```
$ mpiexec -n 8 MSMPIsplitt
'MPI_COMM_WORLD' process rank/size 4/8 has rank/size 2/4 in 'new_comm'
'MPI_COMM_WORLD' process rank/size 6/8 has rank/size 3/4 in 'new_comm'
'MPI_COMM_WORLD' process rank/size 5/8 has rank/size 2/4 in 'new_comm'
'MPI_COMM_WORLD' process rank/size 0/8 has rank/size 0/4 in 'new_comm'
Sum of ranks in 'new_com': 6
'MPI_COMM_WORLD' process rank/size 7/8 has rank/size 3/4 in 'new_comm'
'MPI_COMM_WORLD' process rank/size 1/8 has rank/size 0/4 in 'new_comm'
Sum of ranks in 'new_com': 6
'MPI_COMM_WORLD' process rank/size 3/8 has rank/size 1/4 in 'new_comm'
```

```
'MPI_COMM_WORLD' process rank/size 2/8 has rank/size 1/4 in 'new_comm'
```

The above output confirms our expectations. We have two process groups in the new communicators, each comprising four processes with ranks 0 to 3. Both sums of ranks in in process groups are 6, as expected. □

For an exercise, suppose that we have seven processes in the default communicator `MPI_COMM_WORLD` with initial `ranks` = {0 1 2 3 4 5 6}. Note, that for this case, the program should be executed by `mpiexec` option `-n 7`. Let the `color` be `(rank >= 2)` and `key` be `(rank <= 3)`, which results in process `colors` = {0 0 1 1 1 1 1} and `keys` = {1 1 1 1 0 0 0}. After a call to `MPI_COMM_SPLIT` operation, two process groups are created in `new_comm`, with two and five members, respectively. By using initial `rank` for the processes identification, the processes in new groups are `new_g1` = {0 1} and `new_g2` = {2 3 4 5 6}.

The new ranks of processes in both groups are determined according to corresponding values of keys. Aligning the initial `rank` and `key`, we see for example, that process with initial `rank` = 0 is aligned with `key` = 1, or process with initial `rank` = 4 is aligned with `key` = 0, etc. Now, the keys can be assigned to process groups as: `key_g1` = {1 1} and `key_g2` = {1 1 0 0 0}. Because smaller values of `keys` relate with smaller values of ranks, and because equal `keys` does not change the original rank's order, we get: `rank_g1` = {0 1} and `rank_g2` = {3 4 0 1 2}. For example, process with initial `rank` = 4 becomes a member of `new_g2` with `rank` = 0. Obviously, the sums of ranks in both groups of the new communicator are 1 and 10, respectively. Please, feel free to adapt MPI program `MSMPIsplitt.cpp` from Listing 4.8 in a way that it will implement the described example.

4.8 How effective are your MPI programs?

Already in the simple cases of MPI programs, one can analyse the speed-up as a function of the problem size and as a function of the number of cooperating processes.

The paralization of sequential problems can be guided by various methodologies that provide the same quantitative results, however, in different execution time or with different memory requirements. Some parallelization approaches are better for smaller number of computing nodes and other for larger number of nodes. We are looking for an optimal solution that is simple, efficient and scalable. A simple parallelization methodology, proposed by Ian Foster in his famous book "Designing and Building Parallel Programs", is performed in four distinct stages: Partitioning, Communication, Agglomeration, and Mapping (PCAM).

In the first two stages the sequential problem is decomposed into, as small as possible, tasks and the required communication among the tasks is identified. The available parallel execution platform is ignored for these two phases, because the aim is a maximal decomposition, with the final goal, to improve concurrency and scalability of the discovered parallel algorithms.

The third and fourth stages respect the ability of targeted parallel computer. The identified fine-grained tasks have to be agglomerated to improve performance and to reduce development costs. The last stage is devoted to the mapping of tasks on real computers, taking into account locality of communication and balancing of calculation load.

The developed parallel program speed-up, and consequently, its efficiency, and scalability, depend mainly on the following three issues:

- balancing of computing and communication loads among processes,
- ratio between computing and communication, and
- computer architecture.

Further improvements in the parallelization efficiency could be obtained by an overlapping of calculation with communication, in particular in problems with large messages. Some of approaches to measure the performance of MPI programs are presented in Part III.

4.9 Exercises and mini projects

Test Questions

1. True or false:
   a) MPI is a message-passing library specification not a language or compiler specification.
   b) In the MPI model processes communicate only by shared memory.
   c) MPI is useful for an implementation of MIMD/SPMD parallelism.
   d) A single MPI program is usually written that can run with a general number of processes.
   e) It is necessary to specify explicitly, which part of the MPI code will run with specific processes.
2. True or false:
   a) A group and context together form a communicator.
   b) A default communicator `MPI_COMM_WORLD` contains in its group all initial processes and its context is default.
   c) A process is identified by its rank in the group associated with a communicator.
   d) Maximal rank is equal to size.
3. List, in the required order, a) MPI functions to control starting and terminating procedures of MPI processes.
   b) MPI functions for determining the number of participating processes and the identifier of the current process.
4. Suppose that a process with rank 1 started the execution of `MPI_SEND (buf, 5, MPI_INT, 4, 7, MPI_COMM_WORLD)`.
   a) Which process has to start matching `MPI_RECV` to finish this communication?

b) Write the adequate `MPI_RECV`.
c) What will be received?

5. Name the following definitions of the MPI communication semantics:
   a) An operation may return before its completion, and before the user is allowed to re-use resources (such as buffers) specified in the call.
   b) Return from an operation call indicates that resources can safely be re-used.
   c) A call may require execution of an operation on another process, or communication with another process.
   d) all processes in a group need to invoke the procedure.
6. When a process makes a call to `MPI_RECV`, it will wait patiently until a matching send is posted. If the matching send is never posted, the receive will wait forever.
   a) Name this situation.
   b) Describe a solution to the problem?
7. Give a functional equivalent program segment using non-blocking send to implement blocking MPI send operation: `MPI_SEND`.
8. Name the following definitions of the MPI communication semantics:
   a) If a sender posts two messages to the same receiver, and a receive operation matches both messages, the message first posted will be chosen first.
   b) No matter how long a send has been pending, it can always be overtaken by a message sent from another process.
   c) Does the MPI implementation by itself guarantee fairness?
9. a) Implement a one-to-all MPI broadcast operation whereby a single named process (root) sends the same data to all other processes.
   b) Which process(es) has(have) to call this operation?
10. Suppose an $M \times N$ `array` of doubles stored in a C row-major layout in the sender system memory.
    a) Construct a continuous derived datatype `MPI_newtype` specifying a column of the array.
    b) Write an `MPI_Send` to send the first column of `array`. Try the same for the second column. Note that the first stride starts now at `array[0][1]`.
11. Suppose four processes a, b, c, d, with corresponding oldrank in `comm`: 0, 1, 2, 3. Let `color=oldrank%2` and coresponding `key=` 7, 1, 0, 3. Identify `newgroups` of `newcomm`, sorted by `newranks`, after the execution of:
    `MPI_COMM_SPLIT (comm, color, key, newcomm)`.
12. Which types of parallel program composition are supported by:
    a) `MPI_COMM_DUP (comm, newcomm)` and by
    b) `MPI_COMM_SPLIT (comm, color, key, newcomm)`?
    c) Are the above operations examples of collective operations?

Mini projects

P1 . Implement MPI program for a 2-D finite difference algorithm on a square domain with $n \times n = N$ points. Assume 5 points stencil (actual point and four neighbors). Assume ghost boundary points in order to simplify the calculation

in border points (all stencils, including boundary points, are equal). Compare the obtained results, after specified number of iterations, on a single MPI process and on a parallel multi-core computer, e.g. with up to eight cores. Use the performance models for calculation and communication to explain your results. Plot the execution time as a function of the number of points N and as a function of the number of processes p for, e.g. 10^4 time steps.

P2 . Use MPI point-to-point communication to implement the broadcast and reduce functions. Compare the performance of your implementation with that of the MPI global operations `MPI_BCAST` and `MPI_REDUCE` for different data sizes and different number of processes. Use data sizes up to 10^4 doubles and up to all available number of processes. Plot and explain obtained results.

P3 . Implement the summation of four vectors, each of N doubles, with an algorithm similar to the reduction algorithm. The final sum should be available on all processes. Use four processes. Each of them will initially generate its own vector. Use MPI point-to-point communication to implement your version of the summation of the generated vector. Test your program for small and large vectors. Comment results and compare the performance of your implementation with that of the `MPI_ALLREDUCE`. Explain any differences.

4.10 Bibliographical notes

The primary source of MPI information is available at MPI Forum web site: `https://www.mpi-forum.org/` where the complete MPI library specifications and documents are available. MPI features of Version 2.0 are mostly referenced in this book as later versions include more advanced options, however, they are backwards compatible with MPI 2.0.

Newer MPI standards [10] are trying to better support the scalability in future extreme-scale computing systems using advanced topics as: one-sided communications, extended collective operations, process topologies, external interfaces, etc. Advanced topics, e.g. a virtual shared memory emulation through so called MPI windows, which could simplify the programming and improve the execution efficiency, are beyond the scope of this book and are well covered by continual evolving MPI standard, which should be ultimate reference of enthusiastic programmers.

More demanding readers are adviced to check several well-documented open source references for further reading, e.g. for the MPI standard [16], for MPI implementations [1, 2], and many other internet sources for advanced MPI programming.

Note that besides the parallel algorithm, parallelization methodology [9], and the computational performance of the cooperating computers, the parallel program efficiency depends on the topology and speed of the interconnection network [26].

Chapter 5
OpenCL for massively parallel graphic processors

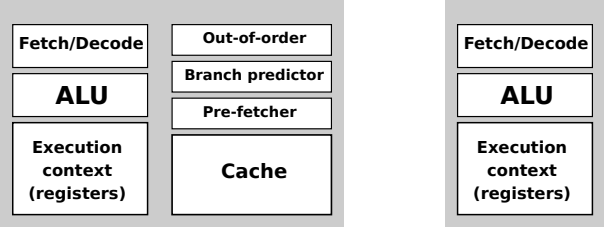
Abstract This chapter aims to provide an introduction to the concepts of parallel programming on GPUs and heterogeneous systems. Almost all desktop computers ship with a quad-core processor and a GPU. Thus we need a programming environment in which a programmer can write programs and run them on either a GPU or a quad-core CPU and a GPU. While central processing units (CPU) are designed to handle complex tasks, such as time slicing, virtual machine emulation, complex control flows and branching, security, etc., graphical processing units (GPUs) only do one thing well. They handle billions of repetitive low level tasks. The first attempts to use GPUs for general-purpose computing were incredibly difficult and took a lot of time and dedication. However, today we have high level languages (such as CUDA and OpenCL) that target the GPUs directly, so GPU programming is rapidly becoming mainstream in the computer science community. In this chapter we will learn how to program GPUs using OpenCL.

5.1 Anatomy of a GPU

In order to understand how to program massively parallel graphic processors, we must first understand how they are built. In the first part of this chapter, we will look behind the idea of processors in graphic processing units (GPU). The basic idea is to have many (hundreds or even thousands) simpler and weaker processing units in GPU instead of one or two powerful CPUs and let these many processors simultaneously perform the same instructions, but with different data. First, let's learn how a GPU is constructed. Then we will learn how we program graphic processing units using the OpenCL language.

Processors in GPU differ from general-purpose CPUs in that they have a much simpler structure that is designed to execute a hundreds of arithmetic instructions simultaneously. To understand how to implement such an efficient massively parallel processor, we will first briefly describe how general-purpose CPUs are built. The simplified structure of a general-purpose single core CPU is presented in

Figure 5.1a. It consists of the instruction fetch and instruction decode logic, an arithmetic-logic unit (ALU) and the execution context. The fetch/decode logic is



(a) Basic structure of a general-purpose single-core CPU.

(b) Basic structure of a slim single-core CPU.

Fig. 5.1: (a) A general-purpose single-core CPU. (b) A slimmed single-core CPU.

responsible for fetching the instructions from memory, decoding them in order to prepare operands and select the required operation in ALU. The execution context comprises the state of CPU such as a program counter, a stack pointer, a program-status register and general purpose registers. Such a general-purpose single core CPU with a single ALU and execution context can run a single instruction from an instruction stream (*thread*) at a time. To increase the performance when executing a single thread, general-purpose single core CPUs rely on out-of-order execution and branch prediction to reduce stalls. However, execution units are of no use without the instructions and the operands, which are stored in main memory. Transferring the instructions and operands to and from main memory requires considerable amount of power and time. This is addressed by the use of caches. Caches work on the principle of either spatial or temporal locality. They work well when an instruction stream is repeated many times (e.g. program loops) and when data is accessed from relatively close memory words. ALUs and fetch/decode logic run at high speed, consume little power and require few hardware resources to build them. Contrary to execution units, a huge number of transistors is needed to build a cache (it may occupy up to 50% of the total die area) and they are very expensive. It is also one of the main energy absorbing element in general-purpose CPU.

5.1.1 Introduction to GPU evolution

To build a GPU that comprises tens or thousands CPUs, we need a slimmer design of a CPU. For this reason all complex and large units should be removed from general-purpose CPU: a branch predictor, out-of order logic, caches and a cache pre-fetcher. Such a single core CPU with a slimmer design is presented in Figure 5.1b.

Now suppose we are running the following fragment of code on a slimmed single-core CPU from Figure 5.1b:

```
1 void vectorAdd( float *vecA, float *vecB, float *vecC ) {
2   int tid = 0;
3   while (tid < 128) {
4     vecC[tid] = vecA[tid] + vecB[tid];
5     tid += 1;
6   }
7 }
```

Listing 5.1: Vector addition

The C code in Listing 5.1 implements vector addition of two floating-point vectors, each containing 128 elements. A slimmed CPU executes a single instruction stream obtained after the compilation of the program in Listing 5.1. A compiled fragment of the function `VectorAdd` that runs on a single core CPU is presented in Figure 5.2. With the first two instructions in Figure 5.2 we clear the registers `r2` and `r3` (suppose

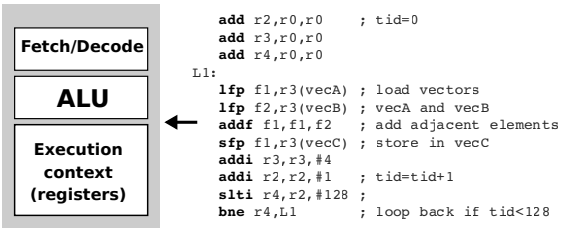


Fig. 5.2: A single instruction stream is executed on a single core CPU.

`r0` iz a zero register). The register `r2` is used to store loop counter (`tid` from Listing 5.1) while the register `r3` contains offset in the vectors `VecA` and `VecB`. Within the `L1` loop CPU loads adjacent elements from the vectors `VecA` and `VecB` into the floating-point registers `f1` and `f2`, adds them and stores the result from the register `f1` into the vector `VecC`. After that we increment the offset in the register `r3`. Recall that the vectors contain floating-point numbers, which are represented with 32 bits (4 bytes), thus the offset is incremented by 4. At the end of the loop we increment the loop counter (variable `tid`) in the register `r2`, compare the loop counter with the value of 128 (the number of elements in each vector) and loop back if the counter is smaller then the length of the vectors `VecA` and `VecB`.

Instead of using one slimmed CPU core from Figure 5.2, we can use two such cores. Why? If we use two CPU cores form Figure 5.2, we will be able to execute two instruction streams fully in parallel (Figure 5.3). A two cores CPU from Figure 5.3 replicates processing resources (Fetch/Decode logic, ALU and execution context) and organizes them into two independent cores. When an application features two instruction streams (i.e. two threads), a two cores CPU provides increased throughput by simultaneously executing these instruction streams on each core. In

the case of vector addition from Listing 5.1 we can now run two threads on each core. In this case each thread will add 64 adjacent vector elements. Notice that both

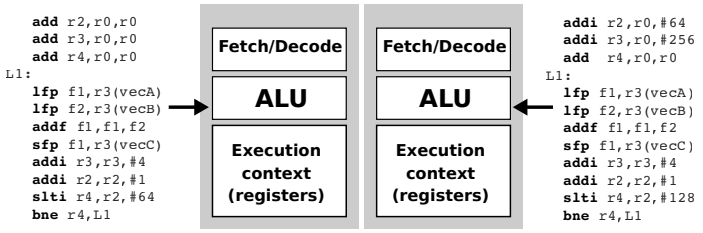


Fig. 5.3: Two instructions streams (two threads) are executed fully in parallel on two CPU cores.

threads in Figure 5.3 have the same instruction stream but use different data. The first thread adds the first 64 elements (the loop index `tid` in the register `r2` iterates from 0 to 63), while the second thread adds the last 64 elements (the loop index `tid` in the register `r2` iterates from 64 to 127).

We can achieve even higher performance by further replicating ALUs and execution contexts as in Figure 5.4. Instead of replicating the complete CPU core from Figure 5.2, we can replicate only ALU and execution context and leaving the fetch/decode logic shared among ALUs. As the fetch/decode logic is shared, all ALUs should execute the same operations contained in an instruction stream, but they can use different input data. Figure 5.4 depicts such a core with eight ALUs, eight execution contexts and shared fetch/decode logic. Such a core usually implements additional storage for data shared among the threads.

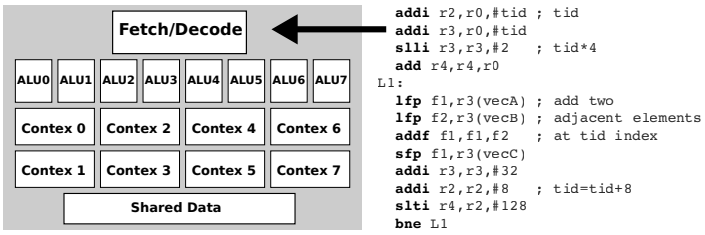


Fig. 5.4: A GPU core with eight ALUs, eight execution contexts and shared fetch/decode logic.

On such a core we can add eight adjacent vector elements in parallel using one instruction stream. The instruction stream is now shared across threads with identical program counters (PC). The same instruction is executed for each thread but on different data. Thus, there is one ALU and one executon context per thread. Each

Compute units and Processing elements

Terminology about processor cores in a modern GPU can be very confusing. The meaning of a term depends on who manufactured a particular GPU. To make thing simple, we opted for the following terminology. A compute unit can be considered equivalent to cores in CPU. Compute units are the basic computational building blocks of GPUs. CPU cores were designed for serial tasks like productivity applications, while GPUs were designed for more parallel and graphics-intensive tasks like video editing, gaming and rich Web browsing. Compute units have many ALUs and execution context that share a common fetch/decode logic. ALUs execute same instructions in a *lock-step basis*, i.e running the same instruction but on different data. These CUs implement an entirely new instruction set that is much simpler for compilers and software developers to use and delivers more consistent performance.

Top two GPU vendors, NVIDIA and AMD, use different names to describe compute units and processing elements. A compute unit is a *stream multiprocessor* in a NVidia GPU or a *SIMD engine* in an AMD GPU. A processing element is a *stream processor* in a NVidia GPU or an *ALU* in an AMD GPU.

thread should now use its own ID (`tid`) to identify data which is to be used in instructions. The compiler for such a CPU core should be able to translate the code from Listing 5.1 into the assembly code from Figure 5.4. When the first instruction is fetched it is dispatched to all 8 ALUs within the core. Recall that each ALU has its own set of registers (execution context) so each ALU would add its own `tid` to its own register `r2`. The same holds also for the second and all following instructions in the instruction stream. For example, the instruction:

```
lfp f1,r3(vecA)
```

is executed on all ALUs at the same time. This instruction loads the element from vector `vecA` at the address `vecA+r3`. Because the value in `r3` is based on different `tid`, each ALU will operate on different element form vector `vecA`. Most modern GPUs use this approach where the cores execute scalar instructions but one instruction stream is shared across many threads.

In this book we well refer to a CPU core from Figure 5.4 as **Compute Unit** (CU) and to ALU as **Processing Element**. Let's summarize the key-features of computer units. We can say that they are general-purpose processors, but they are designed very differently than the general-purpose cores in CPUs: they support so-called SIMD (Single Instruction Muliple Data) parallelism through replication of execution units (ALUs) and corresponding execution contexts, they do not support branch prediction or speculative execution and they have less cache than general-purpose CPUs.

We can further improve the execution speed of our vector addition problem replicating compute units. Figure 5.5 shows a GPU containing 16 compute units. Using 16 compute units as in Figure 5.5 we can add 128 adjacent vector elements in parallel using one instruction stream. Each CU executes a code snippet in Figure 5.5, which represents one thread. Let's suppose that we run 128 threads and each thread

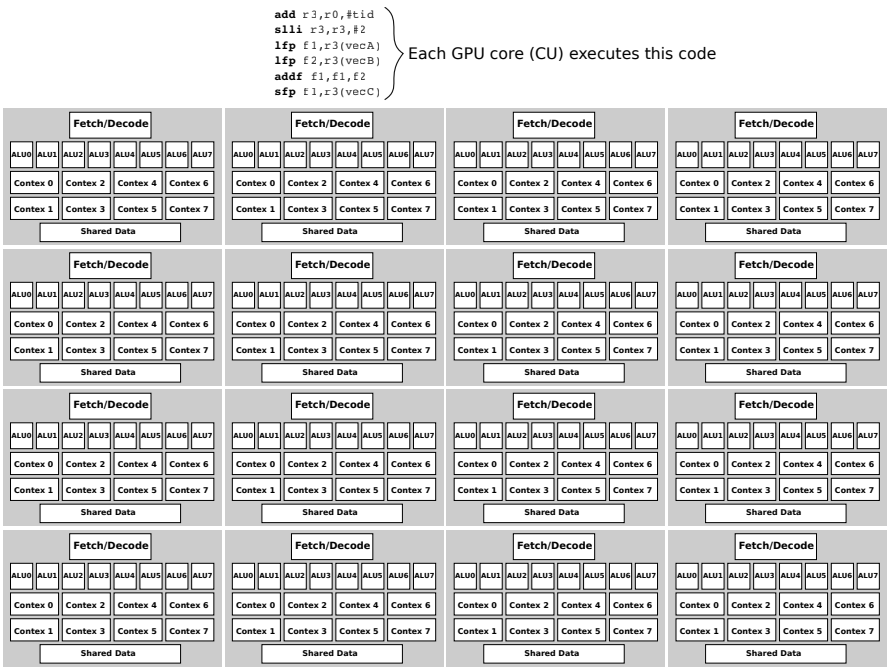


Fig. 5.5: Sixteen streaming multiprocessors each containg eigth execution units and eigth separate contexts.

has its own ID, `tid`, where `tid` is in range $0 \ldots 127$. The first two instructions load the thread ID `tid` into `r3` and multiply it by 4 (in order to obtain the correct offset in floating-point vector). Now the register `r3` that belongs to each thread contains the offset of the vector element that will be accessed in that thread. Each thread then adds two adjacent elements of `vecA` and `vecB` and stores the result into the corresponding element of `vecC`. Because each compute units has 8 processing elements (128 processing elements in total), there is no need for the loop. Hopefully, we are now able to understand the basic idea behind modern GPUs: use as many ALUS as possible and let ALUs execute same instructions in a *lock-step basis*, i.e running the same instruction at the same time but on different data.

5.1.2 A modern GPU

A modern GPUs comprise tens of compute units. The efficiency of wide SIMD processing allows GPUs to pack many CU cores densely with processing elements. For example, the NVIDIA GeForce GTX780 GPU contains 2304 processing elements. These processing elements are organized into 12 CU cores (192 PEs per CU). All

modern GPUs maintain large numbers of execution contexts on chip to provide maximal memory latency-hiding ability. This represents a significant departure from CPU designs, which attempt to avoid or minimize stalls primarily using large, low-latency data caches and complicated out-of-order execution logic. Each CU contains thousands of 32-bit registers that are used to store execution context and are evenly allocated to threads (or PEs). Registers are both the fastest and most plentiful memory in the compute unit. As an example, CU in NVIDIA GeForce GTX780 (Kepler microarchitecture) contains 65,536 (64K) 32-bit registers. To achieve large-scale multithreading, execution contexts must be compact. The number of thread contexts supported by a CU core is limited by the size of on-chip execution context storage. GPUs can manage many thread contexts (and provide maximal latency-hiding ability) when threads use fewer resources. When threads require large amounts of storage, the number of execution contexts (and latency-hiding ability) provided by a GPU drops. Table 5.1 shows the structure of some of the modern NVIDIA GPUs.

Table 5.1: Comparison of NVIDIA GPU generations

	GeForce GTX280	GeForce GTX580	GeForce GTX780
Microarchitecture	Tesla	Fermi	Kepler
CUs	30	16	12
PEs	240	512	2304
PEs per CU	8	32	192
32-bit registers per CU	16K	32K	64K

5.1.3 Scheduling threads on compute units

The GPU device containing hundreds of simple processing elements is ideally suited for computations that can be run in parallel. That is, data parallelism is optimally handled on the GPU device. This typically involves arithmetic on large data sets (such as vectors, matrices and images), where the same operation can be performed across thousands, if not millions, of data elements at the same time. To exploit such a huge parallelism, the programmers should partition their programs into thousands of threads and schedule them among compute units. To make it easier to switch to OpenCL later in this chapter, we will now define and use the same thread terminology as OpenCL does. In that sense, we will use the term **work-item** (WI) for a thread.

Work-items (or threads) are actually scheduled among compute units in two steps:

1. First, a programmer explicitly, within a program, partitions work-items into groups called **work-groups** (WG). A work-group is simply a block of work-

Work-item (WI) and work-group (WG)

A **work-item** in OpenCL is actually a thread in terms of its control flow and its memory model. Work-items are organized into **work-groups**, which are the unit of work scheduled onto compute units. Because of this, work-groups also define the set of work-items that may share data using local memory and may synchronize at barriers.

items that are executed on the same compute unit. Besides that, a work-group also represents a set of work-items that can be synchronized by means of using barriers or memory fences. As a work-group runs on a compute-unit, all work-items within a work-group are able to share local memory that is present within a compute unit (this will be explained in more details in Subsection 5.1.4). After the program has been compiled and sent to execution, the hardware scheduler (which is a part of GPU) evenly assigns work-groups to compute-units. Work-groups execute independently from each other on CUs. If there are more work-groups than CUs, the work-groups are evenly assigned to CUs. Work-groups can be scheduled in any order by the hardware scheduler. In the following sections we will learn how a programmer partitions a program into work-items and work-groups. Figure5.6) shows how a multithreaded program is partitioned into work-groups that are assigned to several CUs.

2. Secondly, the compute unit schedules and executes work-items from the same work-group in groups of 32 parallel work-items called *warps*. When a compute unit is given one or more work-groups to execute, it partitions them into warps and each warp gets scheduled by a warp scheduler for execution. The way a work-group is partitioned into warps is always the same; each warp contains work-items of consecutive, increasing work-items IDs with the first warp containing work-item 0. Individual work-items composing a warp start together at the same program address, but they have their own instruction address counter and register state and are therefore free to branch and execute independently. However, the best performance is achieved when all work-items from the same warp execute the same instructions.

A warp executes one common instruction at a time. It means that work-items in a warp execute in a so-called *lock-step* basis, running the same instruction but on different data. Full efficiency is thus realized when all 32 work-items in a warp execute the same instruction sequence. If work-items in a warp diverge via a conditional branch (i.e. if we use conditional branches within the code executed by work-items), the warp serially executes each branch path taken, disabling work-items that are not on that path. When all branch paths complete, work-items converge back to the same execution path.

At every instruction issue time, a warp scheduler selects a warp that has work-items ready to execute its next instruction, and issues the instruction to those work-items. Work-items that are ready to execute are called **active work-items**. The number of clock cycles it takes for a warp to be ready to execute its next instruction is

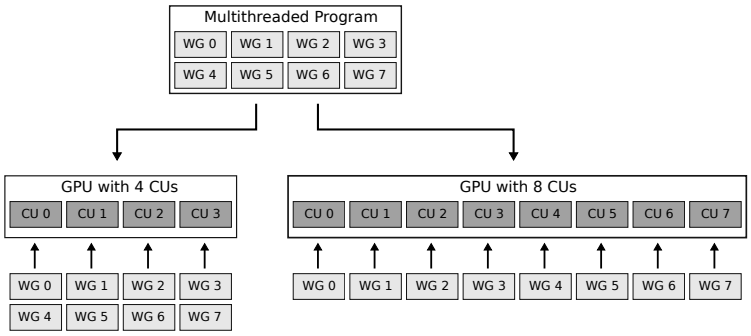


Fig. 5.6: A programmer partitions a program into blocks of *work-item* (threads) called *work-groups*. Work-groups execute independently from each other on CUs. Generally, a GPU with more CUs will execute the program faster than a GPU with fewer CUs.

Warp

A *warp* is a group of 32 work-items from the same work-group that are executed in parallel at the same time. Work-items in a warp execute in a so-called *lock-step* basis. Each warp contains work-items of consecutive, increasing work-items IDs. Individual work-items composing a warp start together at the same program address, but they have their own instruction address counter and register state and are therefore free to branch and execute independently. However, the best performance is achieved when all work-items from the same warp execute the same instructions.

called **latency**. Full utilization is achieved when all warp schedulers have some instruction to issue for some warp at every clock cycle during that latency period. In that case we say that *latency is completely hidden*. The most common reason a warp is not ready to execute its next instruction is that the instruction's input operands are not yet available. Another reason a warp is not ready to execute its next instruction is that it is waiting at some memory fence or barrier. A barrier can force CU to idle as more and more warps wait for other warps in the same work-group to complete execution of instructions prior to the barrier. Full utilization is achieved when more than one work-group is assigned to one CU, so that CU always have 32 work-items from some work-group that are ready to execute and are not waiting at the barrier.

If processing elements within a CU remain idle during the period while a warp is stalled, then a GPU is inefficiently utilized. Instead, GPUs maintain more execution contexts on CU than they can simultaneously execute (recall that a huge register file is used to store context for each work-item). In such a way, PEs can execute instructions from active work-items when others are stalled. The execution context (program counters, registers, etc) for each warp processed by a CU is maintained on-chip during the entire lifetime of the warp. Therefore, switching from one execution

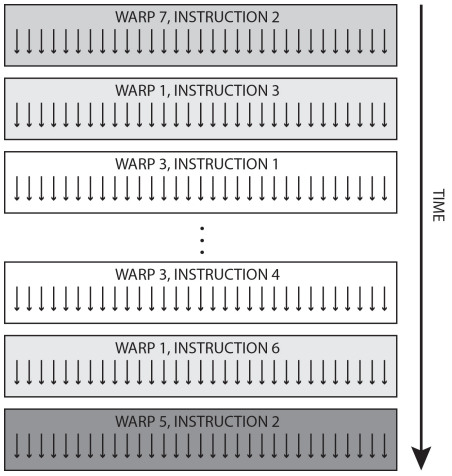


Fig. 5.7: Scheduling of warps within a compute unit. At every instruction issue time, a warp scheduler selects a warp that has work-items ready to execute its next instruction. Each warp always contains work-items of consecutive work-items IDs, but warps are executed out-of-order.

Memory hierarchy on GPU

A GPU device has these five memory regions accessible from a single work-item:

- Registers
- Local Memory
- Texture Memory
- Constant Memory
- Global Memory

context to another has no cost. Also, having multiple resident work-groups per CU can help reduce idling in the case of barriers, as warps from different work-groups do not need to wait for each other at barriers.

5.1.4 Memory hierarchy on GPU

Modern GPUs have several memories that can be accessed from a single work-item. Memory hierarchy of a modern GPU is shown in Figure 5.8. A memory hierarchy has a number of levels of areas where work-items can place data. Each level has its latency (i.e access time) as shown in Table 5.2.

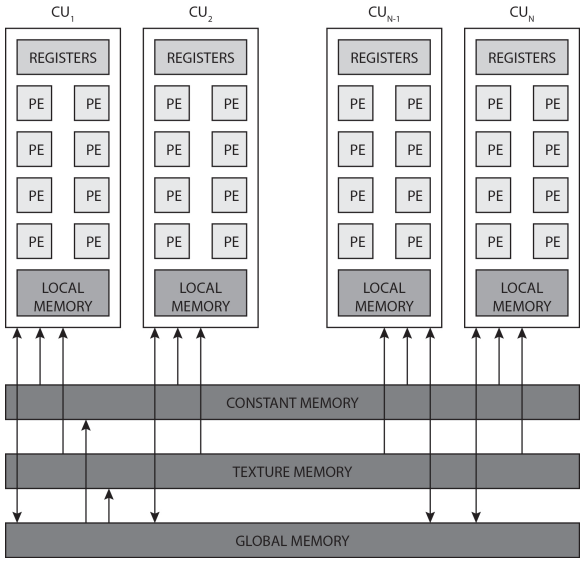


Fig. 5.8: Memory hierarchy on GPU

Table 5.2: Access time by memory level

Storage type	Access time
Registers	1 cycle
Local Memory	1-32 cycles
Texture Memory	~500 cycles
Constant Memory	~500 cycles
Global Memory	~500 cycles

The GPU has thousands of *registers* per compute unit (CU). The registers are at the first and also the most preferable level, as their access time is 1 cycle. Recall that GPU dedicates real registers to each and every work-item. The number of registers per work-item is calculated at compile time. Depending on the particular microarchitecture of a CU, there is 16 K, 32 K or 64 K registers for all work-items within an CU. For example, with Kepler microarchitecture you get 64 K of registers per CU. If you decide to partition your program such that there are 256 work-items per work-group, and that there are four work-groups per CU, you will get 65536 / (256*4) = 64 registers per work-item on a CU.

Each CU contains a small amount (64 kB) of very fast on-chip memory that can be accessed from the work-items running at the particular CU. It is mainly used for data interchange within a work-group running on CU. This memory is called *local or shared memory*. Local memory acts as a user-controlled L1 cache. Actually, on modern GPUs this on-chip memory can be used as a user-controlled

Memory coalescing

Coalesced memory access or *memory coalescing* refers to combining multiple memory accesses into a single transaction. Grouping of work-items into warps is not only relevant to computation, but also to global memory accesses. The GPU device coalesces global memory loads and stores issued by work-items of a warp into as few transactions as possible to minimize DRAM bandwidth. On the recent GPUs, every successive 128 bytes (e.g. 32 single precision words) memory can be accessed by a warp in a single transaction.

local memory or standard hardware-controlled L1 cache. For example, on Kepler CUs this memory can be split of 48 KB local memory / 16 KB L1 cache. On CUs with the Tesla microarchitecture there is 16 kB of local memory and no L1 cache. Local memory has around one-fifth of the speed of registers.

The largest memory space on GPU is the **global memory**. The global memory space is implemented in high-speed GDDR, or graphics dynamic memory, which achieves very high bandwidth, but like all memory, has a high latency. GPU global memory is global because it's accessible from both the GPU and the CPU. It can actually be accessed from any device on the PCI-E bus. For example, the GeForce GTX780 GPU has 3GB of global memory implemented in GDDR5. Global memory resides in device DRAM and it is used for transfers between the host and device as well as for the data input to and output from work-items running on CUs. Reads and writes to global memory are always initiated from CU and are always 128 bytes wide starting at the address aligned at 128-bytes boundary. The blocks of memory that are accessed in one memory transactions are called **segments**. This has an extremely important consequence. If two work-items of the same warp access two data that fall into the same 128-bytes segment, data is delivered in a single transaction. If on the other hand there is data in a segment you fetch that no work-item requested - it is being read anyway and you (probably) waste bandwidth. And if two work-items from the same warp access two data that fall into two different 128-bytes segments, two memory transactions are required. The important thing to remember is that to ensure **memory coalescing** *we want work-items from the same warp to access contiguous elements in memory so to minimize the number of required memory transactions.*

There are also two additional read-only memory spaces within global memory that are accessible by all work-items: **constant memory** and **texture memory**. The constant memory space resides in device memory and is cached. This is where constants and program arguments are stored. Constant memory has two special properties: first, it is cached, and second, it supports broadcasting a single value to all work-items within a warp. This broadcast takes place in just a single cycle. Texture memory is cached so an image read costs one memory read from device memory only on a cache miss, otherwise it just costs one read from the texture cache. The texture cache is optimized for 2D spatial access pattern, so work-items of the

OpenCL kernel

Code that gets executed on a GPU device is called a **kernel** in OpenCL. The kernels are written in a C dialect, which is mostly straightforward C with a lot of built-in functions and additional data types. The body of a kernel function implements the computation to be completed by all work-items.

same warp that read image addresses that are close together will achieve best performance.

5.2 Programmer's view

So far, we have learned how GPUs are built, what are compute units and processing elements, how work-groups and work-items are scheduled on CUs, which memory is present on a modern GPU, and what is the memory hierarchy of a modern GPU. We have mentioned that a programmer is responsible for partitioning programs into work-groups of work-items. In the following sections we will learn what is a programmer's view of a heterogeneous system and how to use OpenCL to program for a GPU.

5.2.1 OpenCL

OpenCL (Open Computing Language) is the open, royalty-free standard for cross-platform, parallel programming of diverse processors found in personal computers, servers, mobile devices and embedded platforms. OpenCL is a framework for writing programs that execute across heterogeneous platforms consisting of central processing units (CPUs), graphics processing units (GPUs) and other types of processors or hardware accelerators. OpenCL specifies:

- programming language for programming these devices, and
- application programming interface to control the platform and execute programs on the compute devices.

OpenCL defines the OpenCL C programming language that is used to write compute **kernels** - the C like functions that implements the task which is to be executed by all work-items running on a GPU. Unfortunately, OpenCL has one significant drawback: it's not easy to learn. Even the most introductory application is difficult for a newcomer to grasp. Prior to jump into OpenCL and take advantage of its parallel-processing capabilities, an OpenCL developer needs to clearly understand

three basic concepts: heterogeneous system (also called platform model), execution model and memory model.

5.2.2 Heterogeneous system

A heterogeneous system (also called platform model) consists of a single *host* connected to one or more OpenCL *devices* (e.g. GPUs, FPGA accelerators, DSP or even CPU). The device is where the OpenCL kernels execute. A typical heterogeneous system is shown in Figure 5.9. An OpenCL program consists of the host program, that runs on the host (typically this is a desktop computer with a general-purpose CPU), and one or more kernels that run on the OpenCL devices. The OpenCL device

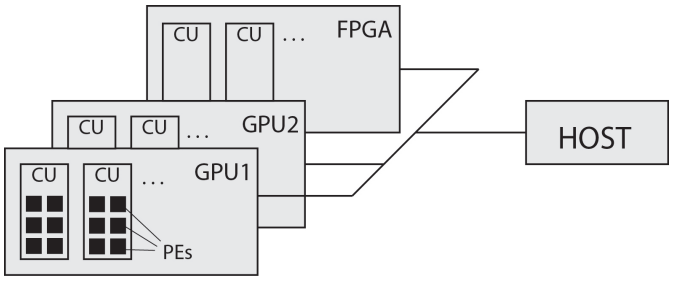


Fig. 5.9: A heterogeneous system.

comprises several compute units. Each compute unit comprises tens or hundreds of processing elements.

5.2.3 Execution model

The OpenCL execution model defines how kernels execute. The most important concept to understand is NDRange (*N-Dimensional Range*) execution. The host program invokes a kernel over an index space. An example of an index space which is easy to understand is a for loop in C. In the for loop defined by the statement `for(int i=0; i<5; i++)`, any statements within this loop will execute five times, with i=0,1,2,3,4. In this case the index space of the loop is [0,1,2,3,4]. In OpenCL, index space is called NDRange, and can have 1, 2, or 3-dimensions. OpenCL kernel functions are executed exactly one time for each point in the NDRange index space. This unit of work for each point in the NDRange is called a work-item. Unlike for loops in C, where loop iterations are executed sequentially and in-order, an OpenCL device is free to execute work-items in parallel and in

OpenCL execution model

The OpenCL Execution Model: Kernels are executed by one or more work-items. Work-items are collected into work-groups and each work-group executes on a compute unit. Kernels are invoked over an index space called NDRange. A work-item is a single kernel instance at a point in the NDRange. NDRange defines the total number of work-items that execute in parallel. In other words, each work-item executes the same kernel function.

any order. Recall that work-items are not scheduled for execution individually onto OpenCL devices. Instead, work-items are organized into work-groups, which are the unit of work scheduled onto compute units. Because of this, work-groups also define the set of work-items that may share data using local memory. Synchronization is possible only between the work-items in a work- group.

Work-items have unique global IDs from the index space. Work-items are further grouped into work-groups and all work-items within a work-group are executed on the same compute unit. Work-groups have a unique work-group ID and work-items have a unique local ID within a work-group. NDRange defines the total number of work-items that execute in parallel. This number is called *global work size* and must be provided by a programmer before the kernel is submitted for execution. The number of work-items within a work-group is called *local work size*. The programmer may also set the local work size at runtime. Work-items within a work-group can communicate with each other and we can synchronize them. In addition, work-items within a work-group are able to share memory. Once the local work size has been determined, the NDRange (global work size) is divided automatically into work-groups, and the work-groups are scheduled for execution on the device.

A kernel function is written on the host. The host program then compiles the kernel and submits the kernel for execution on a device. The host program is thus responsible for creating a collection of work-items, each of which uses the same instruction stream defined by a single kernel. While the instruction stream is the same, each work-item operates on different data. Also, the behavior of each work-item may vary because of branch statements within the instruction stream.

Figure 5.10 shows an example of NDRange where each small square represents a work-item. NDRange in Figure 5.10 is a 2-dimensional index space of size (GX, GY). Each work-item within this NDRange has its own global index (g_x, g_y). For example, the shaded square has global index $(10, 12)$. The work items are grouped into 2-dimensional work-groups. Each work-group contains 64 work-items and is of size (LX, LY). Each work-item within a work-group has a unique local index (l_x, l_y). For example, the shaded square has local index $(2, 4)$. Also, each work-group has its own work-group index (w_x, w_y). For example, the work-group containing the shaded square has work-group index (1,1). And finally, the size of the NDRange index space can be expressed with the number of work-groups in each dimension, (WX, WY).

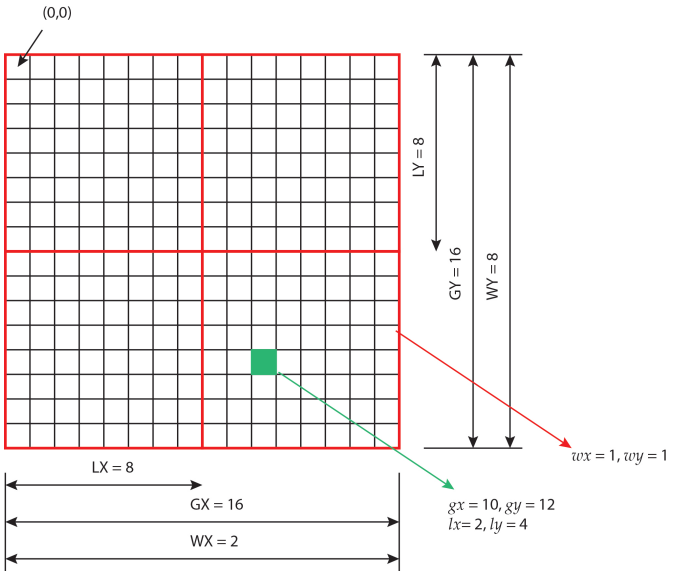


Fig. 5.10: NDRange.

OpenCL memory model

The OpenCL memory model: Kernel data must be specifically placed in one of four address spaces: global memory, constant memory, local memory, or private memory. The location of the data determines how quickly it can be processed and how the data is shared within a work-group.

5.2.4 Memory model

Since common memory address space is unavailable on the host and the OpenCL devices, the OpenCL memory model defines four regions of memory accessible to work-items when executing a kernel. Figure 5.11 shows the regions of memory accessible by the host and the compute device. OpenCL generalizes the different types of memory available on a device into private memory, local memory, global memory and constant memory, as follows:

1. **Private memory**: a memory region that is private per work item. For example, on a GPU device this would be registers within the compute unit.
2. **Local memory**: a memory region that is shared within a work-group. All work-items in the same work-group have both read and write access. On a GPU device this is local memory within the compute unit.

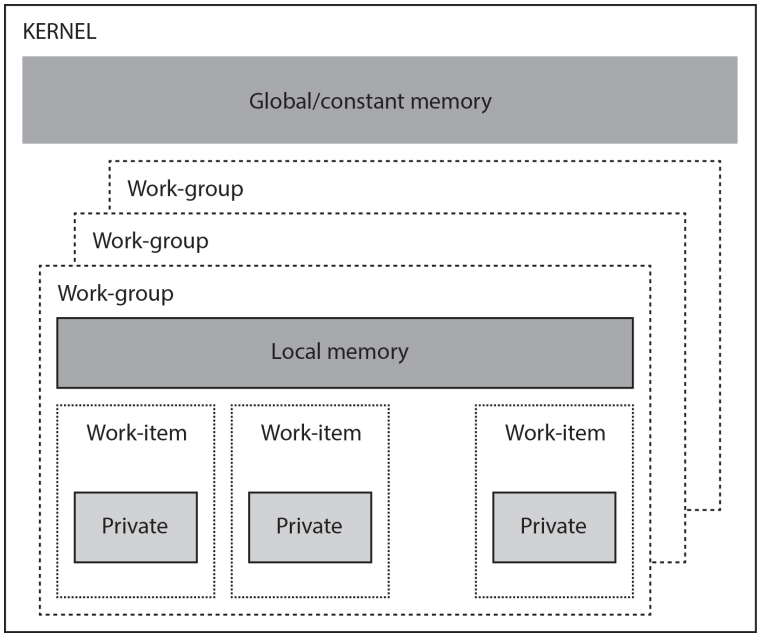


Fig. 5.11: The OpenCL memory model.

3. **Global memory**: a memory region in which all work-items and work-groups have read and write access. It is visible to all work-items and all work-groups. On a GPU device it is implemented in GDDR5. This region of memory can be allocated only by the host during runtime.
4. **Constant memory**: a region of global memory that stays constant throughout the execution of the kernel. Work-items have only read access to this region. The host is permitted both read and write access.

When writing kernels in the OpenCL language, we must declare memory with certain address space qualifiers to indicate whether the data resides in global (`__global`), constant (`__constant`), local (`__local`), or it will default to private within a kernel.

5.3 Programming in OpenCL

5.3.1 A simple example: vector addition

We will start with a simple C program that adds the adjacent elements of two arrays (vectors). with N elements each. The sample C code for vector addition that is intended to run on a single core CPU is shown in Listing 5.2.

```
// add the elements of two arrays
void VectorAdd(float *a,
               float *b,
               float *c,
               int iNumElements) {

    int iGID = 0;

    while (iGID < iNumElements) {
        c[iGID] = a[iGID] + b[iGID];
        iGID += 1;
    }
}
```

Listing 5.2: Sequential vector addition

We compute the sum within a `while` loop. The index `iGID` ranges from 0 to `iNumElements - 1`. In each iteration we add elements `a[iGID]` and `b[iGID]` and place the result in the `c[iGID]`.

Now, we will try to implement the same problem using OpenCL and execute it on a GPU. We will use this simpple problem of adding two vectors because the emphasis will be on getting familiar with OpenCL and not on solving the problem itself. We will show how to split the code into two parts: the *kernel function* and the *host code*.

Kernel function

We can accomplish the same addition on a GPU. To execute the vector addition function on a GPU device, we must write it as a kernel function that is executed on a GPU device. Each thread on the GPU device will then execute the same kernel function. *The main idea is: replace loop iterations with kernel functions executing at each point in a problem domain.* For example, process vectors with `iNumElements` elements with one kernel invocation per element or `iNumElements` threads (kernel executions). The OpenCL kernel is a code sequence that will be executed by every single thread running on a GPU. It is very similar in structure to a C function, but it has the qualifier `__kernel`. This qualifier alerts the compiler that a function is to be compiled to run on an OpenCL device instead the host. The arguments are passed to a kernel as they are passed to any C function. The arguments in the global memory are described with `__global` qualifier and the arguments in the shared memory

OpenCL: Get global ID

The global ID for a working-item in NDRange is obtained by the `get_global_id` function:

```
size_t get_global_id (uint dimindx)
```

This function returns the unique global work-item ID value for dimension identified by its argument `dimindx`. Valid values of `dimindx` are 0 for the first dimension (row), 1 for the second dimension (column) and 2 for the third dimension in NDRange.

are described with `__local` qualifier. These arguments should be always passed as pointers.

As each thread executing the kernel function operates on its own data, there should be a way to identify the thread end link it with particular data. To determine the thread id we use the `get_global_id` function, which works for multiple dimensions.

The kernel function should look similar to the function `VectorAdd` from Listing 5.2. If we assume that each work-item calculates one element of array C, the kernel function looks like in Listing 5.3.

```
// OpenCL Kernel Function for element by element
//  vector addition
__kernel void VectorAdd(
                        __global float* a,
                        __global float* b,
                        __global float* c,
                        int iNumElements
                        ) {

    //find my global index and handle the data at this index
    int iGID = get_global_id(0);

    if (iGID < iNumElements) {
        // add adjacent elements
        c[iGID] = a[iGID] + b[iGID];
    }
}
```

Listing 5.3: Vector Addition - the kernel function

We intend to run this kernel in `iNumElements` instances so that each work-item in NDRange will operate on one vector element. The kernel function has four arguments. The first two arguments are the pointers to input arrays in global memory, `a` and `b`, namely. The third parameter is the pointer to the output array `c` in global memory. and finally, the fourth argument `iNumElements` is the number of elements in arrays. Instead of summing in a `while` loop, each work-item discovers its global index in NDRange and process only the array elements at this index. For one-dimensional arrays we use 1-dimensional index space. To discover its global index in one-dimensional index space, a work-item should call the `get_global_id(0)`

function. Prior to run this kernel on a GPU device, we must setup the execution environment in the host code.

Host code

In developing an OpenCL project, the first step is to code the host application. The host application runs on a user's computer (the host) and dispatches kernels to connected devices. The host application can be coded in C or C++. Because OpenCL supports a wide range of heterogeneous platforms, the programmer must first determine which OpenCL devices are connected to the the platform. After he discovers the devices constituting the platform, the programmer choose one or more devices on which he want to run the kernel function. Only after that can he compile and execute the kernel function on the selected device. Thus, the kernel functions are compiled in runtime and the compilation process is initiated from the host code.

Prior to execute a kernel function, the host program for a heterogeneous system must carry out the following steps:

1. Discover the OpenCL devices that constitute the heterogeneous system. The OpenCL abstraction of the heterogeneous system is represented with *platform* and *devices*. The platform consists of one or more devices capable of executing the OpenCL kernels.
2. Probe the characteristics of these devices so that the software (kernel functions) can adapt to the specific features.
3. Read the program source containing the kernel function(s) and compile the kernel(s) that will run on the selected device(s).
4. Set up memory objects on the selected device(s) that will hold the data for the computation.
5. Compile and run the kernel(s) on the selected device(s).
6. Collect the final result from device(s).

The host code can be very difficult to understand for the beginner, but we will soon realize that a large part of the host code is repeated and can be reused in different applications. Once we understand the host code, we will only devote our attention to writing kernel functions. The above steps are accomplished through the following series of calls to OpenCL API within the host code:

1. Prepare and initialize data on host.
2. Discover and initialize the devices.
3. Create a context.
4. Create a command queue.
5. Create the program object for a context.
6. Build the OpenCL program.
7. Create device buffers.
8. Write host data to device buffers.
9. Create and compile the kernel.
10. Set the kernel arguments.

11. Set the execution model and enqueue the kernel for execution.
12. Read the output buffer back to the host.

Every host application requires five data structures: `cl_device_id`, `cl_context`, *cl_command_queue*, `cl_program` and `cl_kernel`. This data structures must be initialized and filled-in prior to enqueue the kernel function for execution. Listing 5.4 shows the host code. In paragraphs that follows we explain each step within the host code and briefly describe the OpenCL API function used to accomplish the step. For more detailed description of API calls you should refer to OpenCL™ 2.2 Specification.

```
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <OpenCL/opencl.h>

cl_int status;
cl_int ciErr;
cl_device_id *devices = NULL;
cl_uint numDevices = 0;
char buffer[100000];
cl_uint buf_uint;
cl_ulong buf_ulong;
size_t buf_sizet;
cl_int iNumElements = 512*512;

cl_float* srcA;
cl_float* srcB;
cl_float* srcC;
cl_float result;

FILE* programHandle;            // File that contains kernel functions
size_t programSize;
char *programBuffer;
cl_program cpProgram;           // OpenCL program
cl_kernel ckKernel;             // OpenCL kernel

size_t szGlobalWorkSize;        // global work size
size_t szLocalWorkSize;         // local work size

// Main function
// ****************************************************
int main(int argc, char **argv)
{
    // set and log Global and Local work size dimensions
    szLocalWorkSize = 512;
    szGlobalWorkSize = iNumElements;
    // Allocate host arrays
    srcA = (void *)malloc(sizeof(cl_float) * iNumElements);
    srcB = (void *)malloc(sizeof(cl_float) * iNumElements);
    srcC = (void *)malloc(sizeof(cl_float) * iNumElements);
    // init arrays:
    for (int i = 0; i<iNumElements; i++ ) {
        *((cl_float*)srcA + i) = 1.0;
        *((cl_float*)srcB + i) = 1.0;
    }

```

```
52      //*****************************************************
53      // STEP 1: Discover and initialize the devices
54      //*****************************************************
55      // Use clGetDeviceIDs() to retrieve the number of
56      // devices present
57      status = clGetDeviceIDs(
58                              NULL,
59                              CL_DEVICE_TYPE_ALL,
60                              0,
61                              NULL,
62                              &numDevices);
63      if (status != CL_SUCCESS)
64      {
65          printf("Error: Failed to create a device group!\n");
66          return EXIT_FAILURE;
67      }
68
69      printf("The number of devices found = %d \n", numDevices);
70
71      // Allocate enough space for each device
72      devices = (cl_device_id*) malloc(numDevices*sizeof(cl_device_id));
73      // Fill in devices with clGetDeviceIDs()
74      status = clGetDeviceIDs(
75                              NULL,
76                              CL_DEVICE_TYPE_ALL,
77                              numDevices,
78                              devices,
79                              NULL);
80      if (status != CL_SUCCESS)
81      {
82          printf("Error: Failed to create a device group!\n");
83          return EXIT_FAILURE;
84      }
85
86      //*****************************************************
87      // STEP 2: Create a context
88      //*****************************************************
89
90      cl_context context = NULL;
91      // Create a context using clCreateContext() and
92      // associate it with the devices
93      context = clCreateContext(
94                                NULL,
95                                numDevices,
96                                devices,
97                                NULL,
98                                NULL,
99                                &status);
100     if (!context)
101     {
102         printf("Error: Failed to create a compute context!\n");
103         return EXIT_FAILURE;
104     }
105
106     //*****************************************************
107     // STEP 3: Create a command queue
108     //*****************************************************
109     cl_command_queue cmdQueue;
110     // Create a command queue using clCreateCommandQueue(),
111     // and associate it with the device you want to execute
112     // on
113     cmdQueue = clCreateCommandQueue(
114                                     context,
115                                     devices[1], // GPU
116                                     CL_QUEUE_PROFILING_ENABLE,
117                                     &status);
118
```

```
119     if (!cmdQueue)
120     {
121         printf("Error: Failed to create a command commands!\n");
122         return EXIT_FAILURE;
123     }
124 
125     //*************************************************
126     // STEP 4: Create the program object for a context
127     //*************************************************
128     // 4 a: Read the OpenCL kernel from the source file and
129     //      get the size of the kernel source
130     programHandle = fopen("/Users/patriciobulic/FRICL/VectorAdd.cl", "r↩
        ");
131     fseek(programHandle, 0, SEEK_END);
132     programSize = ftell(programHandle);
133     rewind(programHandle);
134 
135     printf("Program size = %lu B \n", programSize);
136 
137     // 4 b: read the kernel source into the buffer programBuffer
138     //      add null-termination-required by clCreateProgramWithSource
139     programBuffer = (char*) malloc(programSize + 1);
140 
141     programBuffer[programSize] = '\0'; // add null-termination
142     fread(programBuffer, sizeof(char), programSize, programHandle);
143     fclose(programHandle);
144 
145     // 4 c: Create the program from the source
146     //
147     cpProgram = clCreateProgramWithSource(
148                     context,
149                     1,
150                     (const char **)&programBuffer,
151                     &programSize,
152                     &ciErr);
153     if (!cpProgram)
154     {
155         printf("Error: Failed to create compute program!\n");
156         return EXIT_FAILURE;
157     }
158     free(programBuffer);
159 
160     //*************************************************
161     // STEP 5: Build the program
162     //*************************************************
163     ciErr = clBuildProgram(
164                             cpProgram,
165                             0,
166                             NULL,
167                             NULL,
168                             NULL,
169                             NULL);
170 
171     if (ciErr != CL_SUCCESS)
172     {
173         size_t len;
174         char buffer[2048];
175 
176         printf("Error: Failed to build program executable!\n");
177         clGetProgramBuildInfo(cpProgram,
178                               devices[1],
179                               CL_PROGRAM_BUILD_LOG,
180                               sizeof(buffer),
181                               buffer,
182                               &len);
183         printf("%s\n", buffer);
184         exit(1);
```

```
185     }
186
187     //*************************************************
188     // STEP 6: Create device buffers
189     //*************************************************
190     cl_mem bufferA; // Input array on the device
191     cl_mem bufferB; // Input array on the device
192     cl_mem bufferC; // Output array on the device
193     //cl_mem noElements;
194
195     // Size of data:
196     size_t datasize = sizeof(cl_float) * iNumElements;
197
198     // Use clCreateBuffer() to create a buffer object (d_A)
199     // that will contain the data from the host array A
200     bufferA = clCreateBuffer(
201                             context,
202                             CL_MEM_READ_ONLY,
203                             datasize,
204                             NULL,
205                             &status);
206
207     // Use clCreateBuffer() to create a buffer object (d_B)
208     // that will contain the data from the host array B
209     bufferB = clCreateBuffer(
210                             context,
211                             CL_MEM_READ_ONLY,
212                             datasize,
213                             NULL,
214                             &status);
215
216     // Use clCreateBuffer() to create a buffer object (d_C)
217     // with enough space to hold the output data
218     bufferC = clCreateBuffer(
219                             context,
220                             CL_MEM_WRITE_ONLY,
221                             datasize,
222                             NULL,
223                             &status);
224
225     //*************************************************
226     // STEP 7: Write host data to device buffers
227     //*************************************************
228     // Use clEnqueueWriteBuffer() to write input array A to
229     // the device buffer bufferA
230     status = clEnqueueWriteBuffer(
231                                 cmdQueue,
232                                 bufferA,
233                                 CL_FALSE,
234                                 0,
235                                 datasize,
236                                 srcA,
237                                 0,
238                                 NULL,
239                                 NULL);
240
241     // Use clEnqueueWriteBuffer() to write input array B to
242     // the device buffer bufferB
243     status = clEnqueueWriteBuffer(
244                                 cmdQueue,
245                                 bufferB,
246                                 CL_FALSE,
247                                 0,
248                                 datasize,
249                                 srcB,
250                                 0,
251                                 NULL,
```

```
                                    NULL);

    //*****************************************************
    // STEP 8: Create and compile the kernel
    //*****************************************************
    // Create the kernel
    ckKernel = clCreateKernel(
                              cpProgram,
                              "VectorAdd",
                              &ciErr);
    if (!ckKernel || ciErr != CL_SUCCESS)
    {
        printf("Error: Failed to create compute kernel!\n");
        exit(1);
    }

    //*****************************************************
    // STEP 9: Set the kernel arguments
    //*****************************************************
    // Set the Argument values
    ciErr = clSetKernelArg(ckKernel,
                           0,
                           sizeof(cl_mem),
                           (void*)&bufferA);
    ciErr |= clSetKernelArg(ckKernel,
                            1,
                            sizeof(cl_mem),
                            (void*)&bufferB);
    ciErr |= clSetKernelArg(ckKernel,
                            2,
                            sizeof(cl_mem),
                            (void*)&bufferC);
    ciErr |= clSetKernelArg(ckKernel,
                            3,
                            sizeof(cl_int),
                            (void*)&iNumElements);

    //*****************************************************
    // Start Core sequence... copy input data to GPU, compute,
    //      copy results back

    //*****************************************************
    // STEP 10: Enqueue the kernel for execution
    //*****************************************************
    // Launch kernel
    ciErr = clEnqueueNDRangeKernel(
                                   cmdQueue,
                                   ckKernel,
                                   1,
                                   NULL,
                                   &szGlobalWorkSize,
                                   &szLocalWorkSize,
                                   0,
                                   NULL,
                                   NULL);
    if (ciErr != CL_SUCCESS)
    {
        printf("Error launchung kernel!\n" );
    }

    // Wait for the command commands to get serviced before
    //  reading back results
    //
    clFinish(cmdQueue);

    //*****************************************************
    // STEP 11: Read the output buffer back to the host
```

```
319     //*************************************************
320     // Synchronous/blocking read of results
321     ciErr = clEnqueueReadBuffer(
322                                 cmdQueue,
323                                 bufferC,
324                                 CL_TRUE,
325                                 0,
326                                 datasize,
327                                 srcC,
328                                 0,
329                                 NULL,
330                                 NULL);
331
332     // Wait for the command commands to get serviced before reading ↩
        back results
333     clFinish(cmdQueue);
334
335     // check the result
336     result = 0.0;
337     for (int i=0; i<iNumElements; i++) {
338         result += srcC[i];
339     }
340     printf("Result = %f \n", result);
341
342     // Cleanup
343     free(srcA);
344     free(srcB);
345     free(srcC);
346
347     if(ckKernel) clReleaseKernel(ckKernel);
348     if(cpProgram) clReleaseProgram(cpProgram);
349     if(cmdQueue) clReleaseCommandQueue(cmdQueue);
350     if(context) clReleaseContext(context);
351
352     if(bufferA) clReleaseMemObject(bufferA);
353     if(bufferB) clReleaseMemObject(bufferB);
354     if(bufferC) clReleaseMemObject(bufferC);
355
356     return 0;
357 }
```

Listing 5.4: Host code for vector addition

1. Discover and initialize the devices

Every OpenCL program requires an OpenCL context, including a list of all OpenCL devices available on the platform. To discover and initialize the devices, we use the `clGetDeviceIDs()` function. We must call the `clGetDeviceIDs()` function for two times. In the first call we use `clGetDeviceIDs()` to retrieve the number of the OpenCL devices present on the platform. The code is shown in Listing 5.5. The number of OpenCL devices is returned in `num_Devices`. Once we know how many OpenCL devices are available on the platform we can obtain the list of all devices available on a platform with the second call of the `clGetDeviceIDs()` function.

The sample code for discovering OpenCL devices is shown in Listing 5.6.

```
1     //*************************************************
2     // STEP 1: Discover and initialize the devices
```

OpenCL: Get device ID

To obtain the list of devices available on a platform we use the `clGetDeviceIDs()` function:

```
cl_int clGetDeviceIDs (
   cl_platform_id platform,
   cl_device_type device_type,
   cl_uint num_entries,
   cl_device_id *devices,
   cl_uint *num_devices)
```

`clGetDeviceIDs` returns CL_SUCCESS if the function is executed successfully. Parameters are:

- `platform`: Refers to the platform ID or can be NULL.
- `device_type`: A bitfield that identifies the type of OpenCL device. Some of valid values are `CL_DEVICE_TYPE_CPU`, `CL_DEVICE_TYPE_GPU` and `CL_DEVICE_TYPE_ALL`. For all other values refer to OpenCL 2.2 Specification.
- `num_entries`: The number of `cl_device` entries that can be added to devices. If devices is not NULL, the `num_entries` must be greater than zero.
- `devices`: A list of OpenCL devices found. In the case that this is NULL, then `clGetDeviceIDs` returns the number of devices in `num_devices`. Otherwise it returns a pointer to the list of available OpenCL devices in `devices`.
- `num_devices`: The number of OpenCL devices available that match `device_type`. If `num_devices` is NULL, this argument is ignored.

Refer to OpenCL™ 2.2 Specification for more detailed description.

```
3     //************************************************
4
5     // Use clGetDeviceIDs() to retrieve the number of
6     // devices present
7     status = clGetDeviceIDs(
8                              NULL,
9                              CL_DEVICE_TYPE_ALL,
10                             0,
11                             NULL,
12                             &numDevices);
13    if (status != CL_SUCCESS)
14    {
15        printf("Error: Failed to create a device group!\n");
16        return EXIT_FAILURE;
17    }
18
19    printf("The number of devices found = %d \n", numDevices);
20
21    // Allocate enough space for each device
22    devices = (cl_device_id*) malloc(numDevices*sizeof(cl_device_id));
23    // Fill in devices with clGetDeviceIDs()
24    status = clGetDeviceIDs(
25                             NULL,
26                             CL_DEVICE_TYPE_ALL,
27                             numDevices,
28                             devices,
29                             NULL);
```

OpenCL: Get device info

To get information about an OpenCL device available on a platform we use the `clGetDeviceInfo()` function:

```
cl_int clGetDeviceInfo(
   cl_device_id device,
   cl_device_info param_name,
   size_t param_value_size,
   void *param_value,
   size_t *param_value_size_ret)
```

clGetDeviceInfo returns CL_SUCCESS if the function is executed successfully. Parameters are:

- `device`: Refers to the device returned by `clGetDeviceIDs`.
- `param_name`: An enumeration constant that identifies the device information being queried. Some of valid values are `CL_DEVICE_MAX_COMPUTE_UNITS`, `CL_DEVICE_MAX_WORK_GROUP_SIZE`, `CL_DEVICE_TYPE`, etc. For all other values refer to OpenCL 2.2 Specification.
- `param_value_size`: Specifies the size in bytes of memory pointed to by `param_value`.
- `param_value`: A pointer to memory location where appropriate values for a given `param_name` as specified in the table below will be returned.Specifies the size in bytes of memory pointed to by `param_value`.
- `param_value_size_ret`: Returns the actual size in bytes of data being queried by `param_value`. If `param_value_size_ret` is NULL, it is ignored.

Refer to OpenCL™ 2.2 Specification for more detailed description.

```
30     if (status != CL_SUCCESS)
31     {
32         printf("Error: Failed to create a device group!\n");
33         return EXIT_FAILURE;
34     }
```

Listing 5.5: Discover and initialize devices

the first call is used to discover the number of present devices. This number is returned in the `numDevices` variable. On an Apple laptop with an Intel GPU there are two discovered devices:

```
The number of devices found = 2
```

Once we know the number of devices, we make enough space in `devices` buffer with `malloc()` and then we make the second call to `clGetDeviceIDs()` to obtain the list of all devices in the `devices` buffer.

We can get and print information about an OpenCL device with the `clGetDeviceInfo()` function.

The sample code for printing information about discovered OpenCL devices is shown in Listing 5.6.

```
printf("=== OpenCL devices: ===\n");
for (int i=0; i<numDevices; i++)
{
    printf("  -- The device with the index %d --\n", i);
    clGetDeviceInfo(devices[i],
                    CL_DEVICE_NAME,
                    sizeof(buffer),
                    buffer,
                    NULL);
    printf("  DEVICE_NAME = %s\n", buffer);
    clGetDeviceInfo(devices[i],
                    CL_DEVICE_VENDOR,
                    sizeof(buffer),
                    buffer,
                    NULL);
    printf("  DEVICE_VENDOR = %s\n", buffer);
    clGetDeviceInfo(devices[i],
                    CL_DEVICE_MAX_COMPUTE_UNITS,
                    sizeof(buf_uint),
                    &buf_uint,
                    NULL);
    printf("  DEVICE_MAX_COMPUTE_UNITS = %u\n",
           (unsigned int)buf_uint);
    clGetDeviceInfo(devices[i],
                    CL_DEVICE_MAX_WORK_GROUP_SIZE,
                    sizeof(buf_sizet),
                    &buf_sizet,
                    NULL);
    printf("  CL_DEVICE_MAX_WORK_GROUP_SIZE = %u\n",
           (unsigned int)buf_uint);
    clGetDeviceInfo(devices[i],
                    CL_DEVICE_MAX_WORK_ITEM_DIMENSIONS,
                    sizeof(buf_uint),
                    &buf_uint,
                    NULL);
    printf("  CL_DEVICE_MAX_WORK_ITEM_DIMENSIONS = %u\n",
           (unsigned int)buf_uint);

    size_t workitem_size[3];
    clGetDeviceInfo(devices[i],
                    CL_DEVICE_MAX_WORK_ITEM_SIZES,
                    sizeof(workitem_size),
                    &workitem_size,
                    NULL);
    printf("  CL_DEVICE_MAX_WORK_ITEM_SIZES = %u, %u, %u \n",
           (unsigned int)workitem_size[0],
           (unsigned int)workitem_size[1],
           (unsigned int)workitem_size[2]);
}
```

Listing 5.6: Print devices' information

The following is the output of the code in Listing 5.6 for an Apple laptop with an Intel GPU:

```
=== OpenCL devices found on platform: ===
  -- Device 0 --
  DEVICE_NAME = Intel(R) Core(TM) i7-4770HQ CPU @ 2.20GHz
  DEVICE_VENDOR = Intel
  DEVICE_MAX_COMPUTE_UNITS = 8
  CL_DEVICE_MAX_WORK_GROUP_SIZE = 2200
  CL_DEVICE_MAX_WORK_ITEM_DIMENSIONS = 3
  CL_DEVICE_MAX_WORK_ITEM_SIZES = 1024, 1, 1
  -- Device 1 --
```

```
DEVICE_NAME = Iris Pro
DEVICE_VENDOR = Intel
DEVICE_MAX_COMPUTE_UNITS = 40
CL_DEVICE_MAX_WORK_GROUP_SIZE = 1200
CL_DEVICE_MAX_WORK_ITEM_DIMENSIONS = 3
CL_DEVICE_MAX_WORK_ITEM_SIZES = 512, 512, 512
```

We can use these information about an OpenCL device later in our host program to automatically adapt the kernel to the specific features. In the above example the device with index 1 is an Intel Iris Pro GPU. It has 40 compute units, each work-group can have up to 1200 work-items, which can span into 3-dimensional NDRange and the maximum size in each dimension is 512. We can also see, that the device with index 0 is an Intel Core i7 CPU, which can also execute OpenCL code. It has 8 compute units (four cores, two hardware threads per core).

2. Create a context

Once we have discovered the available OpenCL devices on the platform and have obtained ad least one device ID, we can create an OpenCL context. The context is used to group devices and memory objects together and to manage command queues, program objects and kernel objects. An OpenCL context is created with one or more devices. Contexts are used by the OpenCL runtime for managing objects such as command-queues, memory, program and kernel objects and for executing kernels on one or more devices specified in the context. An OpenCL context is created with the `clCreateContext` function. Listing 5.7 shows a call to this function.

```
1   //*************************************************
2   // STEP 2: Create a context
3   //*************************************************
4
5   cl_context context = NULL;
6   // Create a context using clCreateContext() and
7   // associate it with the devices
8   context = clCreateContext(
9                             NULL,
10                            numDevices,
11                            devices,
12                            NULL,
13                            NULL,
14                            &status);
15  if (!context)
16  {
17      printf("Error: Failed to create a compute context!\n");
18      return EXIT_FAILURE;
19  }
```

Listing 5.7: Create a context

OpenCL: Create a context

An OpenCL context is created by the `clCreateContext()` function:

```
cl_context clCreateContext(
   cl_context_properties *properties,
   cl_uint num_devices,
   const cl_device_id *devices,
   void *pfn_notify (
      const char *errinfo,
      const void *private_info,
      size_t cb,
      void *user_data
   ),
   void *user_data,
   cl_int *errcode_ret)
```

An OpenCL context is created with one or more devices. Contexts are used by the OpenCL runtime for managing objects such as command-queues, memory, program and kernel objects and for executing kernels on one or more devices specified in the context. `clCreateContext` returns a valid non-zero context and `errcode_ret` is set to CL_SUCCESS if the context is created successfully. Otherwise, it returns NULL value with the values returned in `errcode_ret`. Refer to OpenCL™ 2.2 Specification for more detailed description.

3. Create a command queue

When the context is created, command queues are created that allow commands to be sent to the OpenCL devices associated to the context. Commands are placed into the command queue in order the calls are made. The most common use of queues is to enqueue OpenCL kernel functions for execution on a device. The `clCreateCommandQueue` function is used to create a command queue. By enqueuing commands, we request that the OpenCL device execute the operations in the order. If we have multiple OpenCL devices, we must create a command queue for each OpenCL device and submit commands separately to each. Listing 5.8 shows how to create a command queue for an OpenCL device.

```
1  //*****************************************************
2  // STEP 3: Create a command queue
3  //*****************************************************
4
5  cl_command_queue cmdQueue;
6  // Create a command queue using clCreateCommandQueue(),
7  // and associate it with the device you want to execute
8  // on
9  cmdQueue = clCreateCommandQueue(
10                                 context,
11                                 devices[1], // GPU
12                                 CL_QUEUE_PROFILING_ENABLE,
13                                 &status);
14
```

OpenCL: Create a command queue

To create a command-queue on a specific device use the `clCreateCommandQueue()` function:

```
cl_command_queue clCreateCommandQueue(
   cl_context context,
   cl_device_id device,
   cl_command_queue_properties properties,
   cl_int *errcode_ret)
```

`clCreateCommandQueue` returns a valid non-zero command-queue and `errcode_ret` is set to CL_SUCCESS if the command-queue is created successfully. Otherwise, it returns NULL value with the values returned in `errcode_ret`. The third argument specifies if profiling is enabled (`CL_QUEUE_PROFILING_ENABLE`) to measure execution time of commands or disabled (0). Refer to OpenCL™ 2.2 Specification for more detailed description.

```
15 if (!cmdQueue)
16 {
17     printf("Error: Failed to create a command commands!\n");
18     return EXIT_FAILURE;
19 }
```

Listing 5.8: Create a command queue

4. Create the program object for a context

An OpenCL program consists of a set of kernel functions that are identified as functions declared with the `__kernel` qualifier in the program source. Kernel functions are functions that are executed on a particular OpenCL device. OpenCL programs may also contain auxiliary functions and constant data that can be used by `__kernel` functions.

OpenCL allows applications to create a program object using the program source or binary and build appropriate program executables. This allows applications to determine whether they want to use the pre-built offline binary or load and compile the program source and use the executable compiled/linked online as the program executable. This can be very useful as it allows applications to load and build program executables online on its first instance for appropriate OpenCL devices in the system. These executables can now be queried and cached by the application. Future instances of the application launching will no longer need to compile and build the program executables. The cached executables can be read and loaded by the application, which can help significantly reduce the application initialization time.

To create a program object we use the `clCreateProgramWithSource` function. Listing 5.9 shows how to:

- read the OpenCL kernel from the source file `VectorAdd.cl`,

OpenCL: Create a program object

To create a program object on for a context use the `clCreateProgramWithSource()` function:

```
cl_program clCreateProgramWithSource (
   cl_context context,
   cl_uint count,
   const char **strings,
   const size_t *lengths,
   cl_int *errcode_ret)
```

`clCreateProgramWithSource` creates a program object for a context, and loads the source code specified by the text strings in the strings array into the program object. `clCreateCommandQueue` returns a valid non-zero program object and `errcode_ret` is set to CL_SUCCESS if the program object is created successfully. Otherwise, it returns NULL value with the values returned in `errcode_ret`. Refer to OpenCL™ 2.2 Specification for more detailed description.

- read the kernel source into the buffer `programBuffer`, and
- create the program from the source.

```
1  //*****************************************************
2  // STEP 4: Create the program object for a context
3  //*****************************************************
4
5  // 4 a: Read the OpenCL kernel from the source file and
6  //      get the size of the kernel source
7  programHandle = fopen("/Users/patriciobulic/FRICL/VectorAdd.cl", "r↩
   ");
8  fseek(programHandle, 0, SEEK_END);
9  programSize = ftell(programHandle);
10 rewind(programHandle);
11
12 printf("Program size = %d B \n", programSize);
13
14 // 4 b: read the kernel source into the buffer programBuffer
15 //      add null-termination-required by clCreateProgramWithSource
16 programBuffer = (char*) malloc(programSize + 1);
17
18 programBuffer[programSize] = '\0'; // add null-termination
19 fread(programBuffer, sizeof(char), programSize, programHandle);
20 fclose(programHandle);
21
22 // 4 c: Create the program from the source
23 //
24 cpProgram = clCreateProgramWithSource(
25                context,
26                1,
27                (const char **)&programBuffer,
28                &programSize,
29                &ciErr);
30 if (!cpProgram)
31 {
32     printf("Error: Failed to create compute program!\n");
33     return EXIT_FAILURE;
34 }
```

OpenCL: Build a program executable

To build (compile and link) a program executable from the program source or binary use the `clBuildProgram()` function:

```
cl_int clBuildProgram (
   cl_program program,
   cl_uint num_devices,
   const cl_device_id *device_list,
   const char *options,
   void (*pfn_notify)(cl_program, void *user_data),
   void *user_data)
```

`clBuildProgram` returns CL_SUCCESS if the function is executed successfully. Otherwise, it returns one of errors. Refer to OpenCL™ 2.2 Specification for more detailed description.

```
35
36      free(programBuffer);
```

Listing 5.9: Create the program object for a context

5. Build the program

Once we have created a program object using the function `clCreateProgramWithSource`, we must build a program executable from the contents of the program object. Building the program compiles the source code in the program object and links the compiled code into an executable program. The program object can be built for one or more OpenCL devices using the function `clBuildProgram`. This function builds (compiles and links) a program executable from the program source or binary.The function `clBuildProgram` modifies the program object to include the executable, the build log and build options. Listing 5.10 shows how to build the program and read build information for the selected device in the program object in the case when the build process fails.

```
1       //*****************************************************
2       // STEP 5: Build the program
3       //*****************************************************
4
5       ciErr = clBuildProgram(
6                               cpProgram,
7                               0,
8                               NULL,
9                               NULL,
10                              NULL,
11                              NULL);
12
13      if (ciErr != CL_SUCCESS)
14      {
15          size_t len;
```

```
        char buffer[2048];

        printf("Error: Failed to build program executable!\n");
        clGetProgramBuildInfo(cpProgram,
                              devices[1],
                              CL_PROGRAM_BUILD_LOG,
                              sizeof(buffer),
                              buffer,
                              &len);
        printf("%s\n", buffer);
        exit(1);
    }
```

Listing 5.10: Build the program

6. Create device buffers

Memory objects are reserved regions of global device memory that contains our data. There are two types of memory objects: device buffers and image objects. In this book we use only device buffers. To create a device buffer we use the `clCreateBuffer` function.

One important thing to remember is that we should never try to de-reference the device pointer from the host code as the device memory is not directly accessible from the host, i.e. we cannot use these pointers to buffer objects to read or write memory from code that executes on the host. We use these pointers to read or write memory from code that execute on device. Also , we pass these pointers as arguments to kernels, i.e. functions that execute on device. To read or write to device buffers from the host we must use OpenCL dedicated functions `clEnqueueReadBuffer` and `clEnqueueWriteBuffer`. Upon creation, the contents of the device buffers are undefined. We must explicitly fill the device buffers with our data from the host application. We will show this in the next subsection.

Listing 5.11 shows how to create three device buffers: `bufferA` and `bufferB` that are read-only and are used to store input vectors; and `bufferC` that is write-only and used to store the result of vector addition.

```
    //*****************************************************
    // STEP 6: Create device buffers
    //*****************************************************

    cl_mem bufferA; // Input array on the device
    cl_mem bufferB; // Input array on the device
    cl_mem bufferC; // Output array on the device
    //cl_mem noElements;

    // Size of data:
    size_t datasize = sizeof(cl_float) * iNumElements;

    // Use clCreateBuffer() to create a buffer object (d_A)
    // that will contain the data from the host array A
    bufferA = clCreateBuffer(
                             context,
                             CL_MEM_READ_ONLY,
                             datasize,
```

OpenCL: Create a buffer

To create a device buffer use the `clCreateBuffer` function:

```
cl_mem clCreateBuffer (
   cl_context context,
   cl_mem_flags flags,
   size_t size,
   void *host_ptr,
   cl_int *errcode_ret)
```

This function creates a buffer object within the context `context` of the size `size` bytes using flags `flags`. The pointer to the allocated buffer data `host_ptr` holds the address in the device memory. It returns a valid non-zero buffer object and `errcode_ret` is set to CL_SUCCESS if the program object is created successfully. Otherwise, it returns NULL value with the values returned in `errcode_ret`.
A bit-field `flags` is used to specify allocation and usage information such as the memory arena that should be used to allocate the buffer object and how it will be used. The following are some of the possible values for flags:

CL_MEM_READ_WRITE This flag specifies that the memory object will be read and written by a kernel. This is the default.

CL_MEM_WRITE_ONLY This flags specifies that the memory object will be written but not read by a kernel. Reading from a buffer object created with CL_MEM_WRITE_ONLY inside a kernel is undefined.

CL_MEM_READ_ONLY This flag specifies that the memory object is a read-only memory object when used inside a kernel. Writing to a buffer or image object created with CL_MEM_READ_ONLY inside a kernel is undefined.

Refer to OpenCL™ 2.2 Specification for more detailed description.

```
19                              NULL,
20                              &status);
21
22      // Use clCreateBuffer() to create a buffer object (d_B)
23      // that will contain the data from the host array B
24      bufferB = clCreateBuffer(
25                              context,
26                              CL_MEM_READ_ONLY,
27                              datasize,
28                              NULL,
29                              &status);
30
31      // Use clCreateBuffer() to create a buffer object (d_C)
32      // with enough space to hold the output data
33      bufferC = clCreateBuffer(
34                              context,
35                              CL_MEM_WRITE_ONLY,
36                              datasize,
37                              NULL,
38                              &status);
```

Listing 5.11: Create device buffers

OpenCL: Write to a buffer

To enqueue commands to write to a buffer object from host memory use the `clEnqueueWriteBuffer` function:

```
cl_int clEnqueueWriteBuffer (
   cl_command_queue command_queue,
   cl_mem buffer,
   cl_bool blocking_write,
   size_t offset,
   size_t cb,
   const void *ptr,
   cl_uint num_events_in_wait_list,
   const cl_event *event_wait_list,
   cl_event *event)
```

This function enqueues commands into command queue `command_queue` to write `cb` bytes to a device buffer `buffer` from host memory pointed by `ptr`. This function does not block by default. To know when the command has completed, we can use a blocking form of the command by setting the `blocking_write` parameter to `CL_TRUE`. Refer to OpenCL™ 2.2 Specification for more detailed description.

7. Write host data to device buffers

After we have created the device buffers, we can enqueue reads and writes. To write data from host memory to a device buffer we use the `clEnqueueWriteBuffer` function. We use this function to provide data for processing by a kernel executing on the device. Listing 5.12 shows how to write host data (input vectors `srcA` and `srcB`) to the device buffers `bufferA` and `buffferB`. The device buffers will be then accessed within the kernel.

```
1   //*************************************************
2   // STEP 7: Write host data to device buffers
3   //*************************************************
4   // Use clEnqueueWriteBuffer() to write input array A to
5   // the device buffer bufferA
6   status = clEnqueueWriteBuffer(
7                                 cmdQueue,
8                                 bufferA,
9                                 CL_FALSE,
10                                0,
11                                datasize,
12                                srcA,
13                                0,
14                                NULL,
15                                NULL);
16
17  // Use clEnqueueWriteBuffer() to write input array B to
18  // the device buffer bufferB
19  status = clEnqueueWriteBuffer(
20                                cmdQueue,
21                                bufferB,
22                                CL_FALSE,
```

OpenCL: Create a kernel object

To create a kernel object use the `clCreateKernel` function:

```
cl_kernel clCreateKernel (
        cl_program  program,
         const char *kernel_name,
         cl_int *errcode_ret)
```

This function creates a kernel object from a function `kernel_name` contained within a program object `program` with a successfully built executable. Refer to OpenCL™ 2.2 Specification for more detailed description.

```
                                    0,
                                    datasize,
                                    srcB,
                                    0,
                                    NULL,
                                    NULL);
```

Listing 5.12: Write host data to device buffers

8. Create and compile the kernel

A kernel is a function we declare in an OpenCL program and is executed on the OpenCL device. We must identify kernels with the `__kernel` qualifier to let OpenCL know that the function is a kernel function. The kernel object is created after the executable has been successfully built in the program object. A kernel object is a data structure that includes the kernel function and the data on which the kernel operates. To create a single kernel object we use the `clCreateKernel` function. Before the kernel object is submitted to the command queue for execution, input or output buffers must be provided for any arguments required by the kernel function. If the arguments use device buffers, they must be created first and the data must be explicitly copied into the device buffers. Listing 5.13 shows how to create a kernel object from the kernel function `VectorAdd()`.

```
    //*****************************************************
    // STEP 8: Create and compile the kernel
    //*****************************************************

    // Create the kernel
    ckKernel = clCreateKernel(
                               cpProgram,
                               "VectorAdd",
                               &ciErr);
    if (!ckKernel || ciErr != CL_SUCCESS)
    {
        printf("Error: Failed to create compute kernel!\n");
        exit(1);
```

OpenCL: Set kernel arguments

To set the argument value for a specific argument of a kernel use the `clSetKernelArg` function:

```
cl_int clSetKernelArg (
   cl_kernel kernel,
   cl_uint arg_index,
   size_t arg_size,
   const void *arg_value)
```

Arguments to the kernel are referred by indices that go from 0 for the leftmost argument to $n-1$, where n is the total number of arguments declared by a kernel. The argument `index` refers to the specific argument position of the kernel definition that must be set. The last two arguments of `clSetKernelArg` specify the size of the argument data and the pointer to the actual data that should be used as the argument value. If a kernel function argument is declared to be a pointer of a built-in or user defined type with the __global or __constant qualifier, a buffer memory object must be used. Refer to OpenCL™ 2.2 Specification for more detailed description.

```
14      }
```

Listing 5.13: Create and compile the kernel

9. Set the kernel arguments

Prior to enqueue the kernel function for execution on device, we must set the kernel arguments. When the required memory objects have been successfully created, kernel arguments can be set using the `clSetKernelArg` function. Listing 5.14 shows how to set arguments for the kernel function `VectorAdd()`. In this example, the input arguments have indices 0, 1, and 3 and the output argument has index 2.

```
1       //*****************************************************
2       // STEP 9: Set the kernel arguments
3       //*****************************************************
4       // Set the Argument values
5       ciErr = clSetKernelArg(ckKernel,
6                              0,
7                              sizeof(cl_mem),
8                              (void*)&bufferA);
9       ciErr |= clSetKernelArg(ckKernel,
10                              1,
11                              sizeof(cl_mem),
12                              (void*)&bufferB);
13      ciErr |= clSetKernelArg(ckKernel,
14                              2,
15                              sizeof(cl_mem),
16                              (void*)&bufferC);
17      ciErr |= clSetKernelArg(ckKernel,
18                              3,
```

```
                                sizeof(cl_int),
                                (void*)&iNumElements);
```

Listing 5.14: Set the kernel arguments

10. Enqueue the kernel for execution

OpenCL always executes kernels in parallel, i.e. instances of the same kernel execute on different data set. Each kernel execution in OpenCL is called a work-item. Each work-item is responsible for executing the kernel once and operating on its assigned portion of the data set. OpenCL exploits parallel computation of the compute devices by having instances of the kernel execute on different portions of the N-dimensional problem space. In addition, each work-item is executed only with its assigned data. Thus, it is programmer's responsibility to tell OpenCL how many work-items are needed to process all data.

Before the work-items total can be determined, the N-dimension to be used to represent the data must be determined. For example, a linear array of data would be a one-dimension problem space, while an image would be a two-dimensional problem space, and spatial data, such as a 3D object, would be a three-dimensional problem space.

When the dimension space is determined, the total work-items (also called the global work size) and group size can be calculated. When the work-items for each dimension and the group size (local work size) is determined (i.e. NDRange), the kernel can be sent to the command queue for execution. To execute the kernel function, we must enqueue the kernel object into a command queue. To enqueue the kernel to execute on a device, we use the function `clEnqueueNDRangeKernel`. Listing 5.15 shows how to enqueue the `VectorAdd` kernel on a device over 1-dimensional space. The total number of work-items is `szGlobalWorkSize`, and the work-group size is `szLlobalWorkSize`. In In this example (vector addition), `szGlobalWorkSize` is set to be equal to the number of elements in the input vector(s), while `szLocalWorkSize` is set to 256. Thus, the kernel `VectorAdd` will be executed by `szGlobalWorkSize` work-items and each SM on a GPU device will execute 256 work-items.

```
    //*************************************************
    // Start Core sequence... copy input data to GPU, compute,
    //     copy results back

    //*************************************************
    // STEP 10: Enqueue the kernel for execution
    //*************************************************
    // Launch kernel
    ciErr = clEnqueueNDRangeKernel(
                                    cmdQueue,
                                    ckKernel,
                                    1,
                                    NULL,
                                    &szGlobalWorkSize,
                                    &szLocalWorkSize,
```

OpenCL: Enqueue the kernel for execution on a device

To enqueue a command to execute a kernel on a device use the function `clEnqueueNDRangeKernel`:

```
cl_int clEnqueueNDRangeKernel (
   cl_command_queue command_queue,
   cl_kernel kernel,
   cl_uint work_dim,
   const size_t *global_work_offset,
   const size_t *global_work_size,
   const size_t *local_work_size,
   cl_uint num_events_in_wait_list,
   const cl_event *event_wait_list,
   cl_event *event)
```

This function enqueues a command into `command_queue` to execute the kernel `kernel` on a device over NDRange. The argument `work_dim` denotes the number of dimensions used to specify the global work-items and work-items in the work-group. The number of global work-items in `work_dim` dimensions that will execute the kernel function is specified with `global_work_size`.The size of the work-group that will execute the kernel is specified with `local_work_size`. Refer to OpenCL™ 2.2 Specification for more detailed description.

```
16                                      0,
17                                      NULL,
18                                      NULL);
19      if (ciErr != CL_SUCCESS)
20      {
21          printf("Error launchung kernel!\n" );
22      }
23
24      // Wait for the command commands to get serviced before
25      //  reading back results
26      //
27      clFinish(cmdQueue);
```

Listing 5.15: Enqueue the kernel for execution

11. Read the output buffer back to the host

After the kernel function has been executed on the device, we shuld read the output data from the device. To read data from a device buffer to host memory we use the `clEnqueueReadBuffer` function. Listing 5.16 shows how to read data from the device buffer `bufferC` to host memory `srcC`.

```
1       //*************************************************
2       // STEP 11: Read the output buffer back to the host
3       //*************************************************
4
```

OpenCL: Read from a buffer

To enqueue commands to read from a buffer object to host memory use the `clEnqueueReadBuffer` function:

```
cl_int clEnqueueReadBuffer (
   cl_command_queue command_queue,
   cl_mem buffer,
   cl_bool blocking_read,
   size_t offset,
   size_t cb,
   void *ptr,
   cl_uint num_events_in_wait_list,
   const cl_event *event_wait_list,
   cl_event *event)
```

This function enqueues commands into command queue `command_queue` to read `cb` bytes from a device buffer `buffer` to host memory pointed by `ptr`. This function does not block by default. To know when the command has completed, we can use a blocking form of the command by setting the `blocking_write` parameter to `CL_TRUE`. Refer to OpenCL™ 2.2 Specification for more detailed description.

```
5     // Synchronous/blocking read of results
6     ciErr = clEnqueueReadBuffer(
7                                 cmdQueue,
8                                 bufferC,
9                                 CL_TRUE,
10                                0,
11                                datasize,
12                                srcC,
13                                0,
14                                NULL,
15                                NULL);
```

Listing 5.16: Read the output buffer back to the host

5.3.2 Sum of arbitrary long vectors

The OpenCL standard does not specify how the abstract execution model provided by OpenCL is mapped to the hardware. We can enqueue any number of threads (work items), and provide a work-group size (number of work_items in a work-group), with at least the following constraints:

1. work-group size must divide the number of work items,
2. work-group size be at most `CL_DEVICE_MAX_WORK_GROUP_SIZE` (recall that for the CPU device used in the previous example this is 1200).

Maximum number of work-groups per compute unit is limited by the hardware resources. Each compute unit has a limited number of registers and a limited amount

Occupancy

Occupancy is a ratio of active warps per compute unit to the maximum number of allowed warps. We should always keeping the occupancy high, because this is a way to hide latency when executing instructions. A compute unit should have a warp ready to execute in every cycle as this is the only way to keep hardware busy.

of local memory. Usually no more than 16 work-groups can run simultaneously on a single compute unit with the Kepler microarchitecture and 8 work-groups can run simultaneously on a single compute unit with the Fermi microarchitecture . Also there is a limit of the number of active warps on a single compute unit (64 on Kepler, 48 on Fermi). We usually want to keep as many active warps as possible, because this affects *occupancy*. Occupancy is a ratio of active warps per compute unit to the maximum number of allowed warps. Keeping the occupancy high, we can hide latency when executing instructions. Recall that executing other warps when one warp is paused is the only way to hide latency and keep hardware busy. Finally, the hardware also limits the number of work-groups in a single launch (usually this is 65535 in each NDRange dimension).

In our previous example we have vectors with 512×512 elements (`iNumElements`) and we have launched the same number of work-items (`szGlobalWorkSize`). As each work-group contains 512 work-items (`szLocalWorkSize`), we have 512 work-groups in a single launch. Is it possible to add larger vectors and where is the limit? If we tried to add two vectors with 512×512 elements, we would fail to launch a kernel with such a large number of work-items (work-groups). So how would we use a GPU to add two arbitrary long vectors? First, we should limit the number of work-items and the number of work-groups. Secondly, one work-item should perform more than one addition. Lets first look at the new kernel function in Listing 5.17.

```
// OpenCL Kernel Function for element by element
//  vector addition of arbitrary long vectors
__kernel void VectorAddArbitrary(
                        __global float* a,
                        __global float* b,
                        __global float* c,
                        int iNumElements
                        ) {

    //find my global index
    int iGID = get_global_id(0);

    while (iGID < iNumElements) {
        // add adjacent elements
        c[iGID] = a[iGID] + b[iGID];
        iGID += get_global_size(0);
    }
}
```

Listing 5.17: Sum o arbitrary long vectors - the kernel function

We used a `while` loop to iterate through the data (this kernel is very similar to the function from Listing 5.2). Rather than incrementinc `iGID` by 1, a many core GPU device could increment `iGID` by the number of work-items that we are using. We want each work-item to start on a different data index, so we use the thread global index:

```
int iGID = get_global_id(0);
```

After each thread finishes its work at the current index, we increment `iGID` by the total number of work-items in NDRange. This number is obtained from the function `get_global_size(0)`:

```
iGID += get_global_size(0);
```

The only remaining piece is to fix the execution model in the host code. To ensure that we never launch too many work-groups and work-items, we will fix the number of work-groups to a small number, but still large enough to have a good occupancy. We will launch 512 work-groups with 256 work-items per work-group (thus the total number of work-items will be 131072). The only change in the host code is:

```
szLocalWorkSize = 256;
szGlobalWorkSize = 512*256;
```

5.3.3 Dot product in OpenCL

We will now take a look at vector dot products. We will start with the simple version first to illustrate basic features of memory and work-item management in OpenCL programs. We will again recap the usage of NDRange and work-item ID. We will than analyze performance of the simple version and extend the simple version to version which employs local memory.

The computation of a vector dot product consists of two steps. First, we multiply corresponding elements of the two input vectors. This is very similar to vector addition but utilizes multiplication instead of addition. In the second step we sum all the products instead of just storing them to an output vector. Each working-item multiplies a pair of corresponding vector elements and then moves on to its next pair. Because the result would be be the sum of all these pairwise products, each working-item keeps a sum of its products. Just like in the addition example, the working-items increment their indices by the total number of threads. The kernel function for the first step is shown in Listing 5.18.

```
1 // OpenCL Kernel Function for Naive Dot Product
2 __kernel void DotProductNaive(
3                                 __global float* a,
4                                 __global float* b,
5                                 __global float* c,
6                                 int iNumElements
7                                 ) {
```

```
8
9       //find my global index
10      int iGID = get_global_id(0);
11      int index = iGID;
12
13      while (iGID < iNumElements) {
14          // add adjacent elements
15          c[iGID] = a[index] * b[index];
16          index += get_global_size(0);
17      }
18  }
```

Listing 5.18: Vector Dot Product Kernel - naive implementation

Each element of the array `c` holds the sum of products obtained form one work-item, i.e. `c[iGID]` holds a sum of products obtained by the work-item with the global index `iGID`. After all work-item finish their work, we should sum a the elements form the vector `c` to produce a single scalar product. But how do we know when have all work-items finished their work? We need a mechanism to synchronize work-items. The only way to synchronize all work-items in NDRange is to wait for the kernel function to finish. After the kernel function finishes, we can read the results (vector `c`) from a GPU device and sum its elements on host. the host code is very similar to the host code from Listing 5.4. We have to make two changes:

1. the vector `c` has a different size than vectors `a` and `b`. It has the same number of elements as the number of all work-items in NDRange (`szGlobalWorkSize`):

```
1       //*************************************************
2       // STEP 6: Create device buffers
3       //*************************************************
4
5       ...
6
7       cl_mem bufferC; // Output array on the device
8
9       // Size of data for bufferC:
10      size_t datasize_c = sizeof(cl_float) * szGlobalWorkSize;
11
12      ...
13
14      // Use clCreateBuffer() to create a buffer object (d_C)
15      // with enough space to hold the output data
16      bufferC = clCreateBuffer(
17                                context,
18                                CL_MEM_READ_WRITE,
19                                datasize_c,
20                                NULL,
21                                &status);
```

Listing 5.19: Create a buffer object C for naive implementation of vector dot product

2. after the kernel executes on a GPU device, we read vector `c` from device and serially sum all its elements to produce the final dot product:

OpenCL: Timing the execution

To return profiling information for the command associated with event if profiling is enabled use the function `clGetEventProfilingInfo`.

```
cl_int clGetEventProfilingInfo (
   cl_event event,
   cl_profiling_info param_name,
   size_t param_value_size,
   void *param_value,
   size_t *param_value_size_ret)
```

The function returns profiling information for the command associated with event if profiling is enabled. The first argument is the event being queried, and the second argument, `param_name` is an enumeration value describing the query. Most often used `param_name` values are:

CL_PROFILING_COMMAND_START : A 64-bit value that describes the current device time counter in nanoseconds when the command identified by event starts execution on the device, and

CL_PROFILING_COMMAND_END : A 64-bit value that describes the current device time counter in nanoseconds when the command identified by event has finished execution on the device.

Event objects are used to capture profiling information that measure execution time of a command. Profiling of OpenCL commands can be enabled using a command-queue created with CL_QUEUE_PROFILING_ENABLE flag set in properties argument to `clCreateCommandQueue`. OpenCL devices are required to correctly track time across changes in device frequency and power states. Refer to OpenCL™ 2.2 Specification for more detailed description.

How fast is your OpenCL kernel

Our motivation for writing kernels in OpenCL is to speed up applications. Often we want to measure the execution time of an kernel. As OpenCL is a performance oriented language, performance analysis is an essential part of OpenCL programming. The OpenCL runtime provides a built-in mechanism (**profiling**) for timing the execution of kernels. A profiler is a performance analysis tool that gathers data from the OpenCL run-time using events. This information is used to discover bottlenecks in the application and find ways to optimize the application's performance. OpenCL supports 64-bit timing of commands submitted to command queues and events to keep track of a command's status. Events can be used with most commands placed on the command queue: commands to read, write, map or copy memory objects, commands to enqueue kernels, etc. Profiling is enabled when a queue is created with the `CL_QUEUE_PROFILING_ENABLE` flag is set. The fact is that, when you execute your kernels on GPU, no CPU clock is spent during the execution. When profiling is enabled, the function `clGetEventProfilingInfo` is used to extract the timing data. We need to follow next steps to measure the execution time of OpenCL kernel

execution time:

1. Create a queue, profiling need to be enabled when the queue is created.

```
cmdQueue = clCreateCommandQueue(
             ...
             CL_QUEUE_PROFILING_ENABLE,
             &status);
```

2. Link an event when launching a kernel:

```
cl_event kernelevent;
ciErr = clEnqueueNDRangeKernel(
          ...
          &kernelevent);
```

3. Wait for the kernel to finish:

```
ciErr = clWaitForEvents (1, &kernelevent); // Wait for the event
```

4. Get profiling data using the function `clGetEventProfilingInfo()` and calculate the kernel execution time.
5. Release the event using the function `clReleaseEvent()`.

The code snippet form Listing 5.20 shows how to measure kernel execution time using OpenCL profiling events.

```
1
2  ...
3
4  //*************************************************
5  // STEP 3: Create a command queue
6  //*************************************************
7
8  cl_command_queue cmdQueue;
9  // Create a command queue using clCreateCommandQueue(),
10 // and associate it with the device you want to execute
11 // on. Enable profiling.
12 cmdQueue = clCreateCommandQueue(
13                                  context,
14                                  devices[1], // GPU
15                                  CL_QUEUE_PROFILING_ENABLE,
16                                  &status);
17
18
19 ...
20
21
22 //*************************************************
23 // Start Core sequence... copy input data to GPU, compute,
24 //     copy results back
25 cl_event kernelevent;
26 //*************************************************
27 // STEP 10: Enqueue the kernel for execution
28 //*************************************************
29 // Launch kernel
30 ciErr = clEnqueueNDRangeKernel(
31                                 cmdQueue,
32                                 ckKernel,
33                                 1,
34                                 NULL,
```

```
                                &szGlobalWorkSize,
                                &szLocalWorkSize,
                                0,
                                NULL,
                                &kernelevent);

if (ciErr != CL_SUCCESS)
{
    printf("Error launchung kernel!\n" );
}
ciErr = clWaitForEvents (1, &kernelevent); // Wait for the event

// Obtain the start- and end time for the event
unsigned long start = 0;
unsigned long end = 0;

// read device time counter in nanoseconds when the command
// identified by event starts execution on the device:
clGetEventProfilingInfo(kernelevent,
                        CL_PROFILING_COMMAND_START,
                        sizeof(cl_ulong),
                        &start,
                        NULL);
clGetEventProfilingInfo(kernelevent,
                        CL_PROFILING_COMMAND_END,
                        sizeof(cl_ulong),
                        &end,
                        NULL);

// Compute the duration in nanoseconds
float duration = (end - start) * 10e-9;

// Don't forget to release the event
clReleaseEvent(kernelevent);

printf("Kernel execution time = %f s \n", duration);

// Wait for the command commands to get serviced before
//  reading back results
//
clFinish(cmdQueue);
```

Listing 5.20: Measuring kernel execution time

This way we can profile operations on both memory objects and kernels. Results for dot product of two vectors of size 16777216 ($512 \times 512 \times 64$) on an Apple laptop with an Intel GPU are:

```
Kernel execution time = 0.127389 s
Result = 33554432.000000
```

5.3.4 Dot product in OpenCL using local memory

Host device data transfer has much lower bandwidth than global memory access. So we should perform as much computation on a GPU device as possible and read as small amount of data from a GPU device as possible. In this case the threads should cooperate to calculate the final sum. Wotk-items can safely cooperate through local memory by means of synchronization. Local memory can be shared by all work-

items in a work-group. Local memory on a GPU is implemented on a compute device. To allocate slocal memory, the `__local` address space qualifier is used in variable declaration. We will use a buffer in local memory named `ProductsWG` to store each work-item's running sum. This buffer will store `szLocalWorkSize` products so each work-item in the work-group will have a place to store its temporary result. Since the compiler will create a copy of the local variables for each work-group, we need to allocate only enough memory such that each thread in the work-group has an entry. It's relatively simple to declare local memory buffers as we just pass local arrays as arguments to the kernel:

```
__kernel void DotProductShared(
                __global float* a,
                __global float* b,
                __global float* c,
                __local* ProductsWG,
                int iNumElements)
```

We then set the kernel argument with a value of `NULL` and a size equal to the size we want to allocate for the argument (in byte). Therefore it should be:

```
ciErr |= clSetKernelArg(ckKernel,
                        3,
                        sizeof(float) * szLocalWorkSize,
                        NULL);
```

Now each work-item computes a running sum of the product of corresponding entries in `a` and `b`. After reaching the end of the array, each thread stores its temporary sum into the local memory (buffer `ProductsWG`):

```
// work-item global index
int iGID = get_global_id(0);
// work-item local index
int iLID = get_local_id(0);
float temp = 0.0;
while (iGID < iNumElements) {
   // multiply adjacent elements
   temp += a[iGID] * b[iGID];
   iGID += get_global_size(0);
}
//store the product in local memory
ProductsWG[iLID] = temp;
```

At this point, we need to sum all the temporary values we have placed in the `ProductsWG`. To do this, we will need some of the threads to read the values that have been stored there. This is a potentially dangerous operation. We should place a synchronization barrier to guarantee that all of these writes to the local buffer `ProductsWG` complete before anyone tries to read from this buffer. The OpenCL C language provides functions to allow synchronization of work-items. However, as we mentioned, the synchronization can only occur between work-items in the same work-group. To achieve that, OpenCL implements a barrier memory fence for synchronization with the `barrier()` function. The function `barrier()` creates a barrier that blocks the current work-item until all other work-items in the same group has executed the barrier before allowing the work-item to proceed beyond

OpenCL: Barrier

```
\texttt{barrier(mem_fence_flag)}
```

All work-items in a work-group executing the kernel on a processor must execute this function before any are allowed to continue execution beyond the barrier. The `mem_fence_flag` can be either `CLK_LOCAL_MEM_FENCE` (the barrier function will queue a memory fence to ensure correct ordering of memory operations to local memory), or `CLK_GLOBAL_MEM_FENCE` (the barrier function will queue a memory fence to ensure correct ordering of memory operations to global memory. This can be useful when work-items, for example, write to buffer objects and then want to read the updated data). Refer to OpenCL™ 2.2 Specification for more detailed description.

the barrier. All work-items in a work-group executing the kernel on a processor must execute this function before any are allowed to continue execution beyond the barrier. The following call guarantees that every work-item in the work-group has completed instructions before the hardware will execute the next instruction on any work-item within the work-group

```
barrier(CLK_LOCAL_MEM_FENCE);
```

Now that we have guaranteed that our local memory has been filled, we can sum the values in it. We call the general process of taking an input array and performing some computations that produce a smaller array of results a *reduction*. The naive way to accomplish this reduction would be having one thread iterate over the shared memory and calculate a running sum. This will take us time proportional to the length of the array. However, since we have hundreds of threads available to do our work, we can do this reduction in parallel and take time that is proportional to the logarithm of the length of the array. Figure 5.12 shows a summation reduction. The idea is that each work-item adds two of the values in `ProductsWG` and store the result back to `ProductsWG`. Since each thread combines two entries into one, we complete the first step with half as many entries as we started with. In the next step, we do the same thing on the remaining half. We continue until we have the sum of every entry in the first element of `ProductsWG`. The code for the summation reduction is:

```
// how many work-items are in WG?
int iWGS = get_local_size(0);
// Summation reduction:
int i = iWGS/2;
while(i!=0){
   if (iLID < i) {
      ProductsWG[iLID] += ProductsWG[iLID+i];
   }
   barrier(CLK_LOCAL_MEM_FENCE);
   i=i/2;
}
```

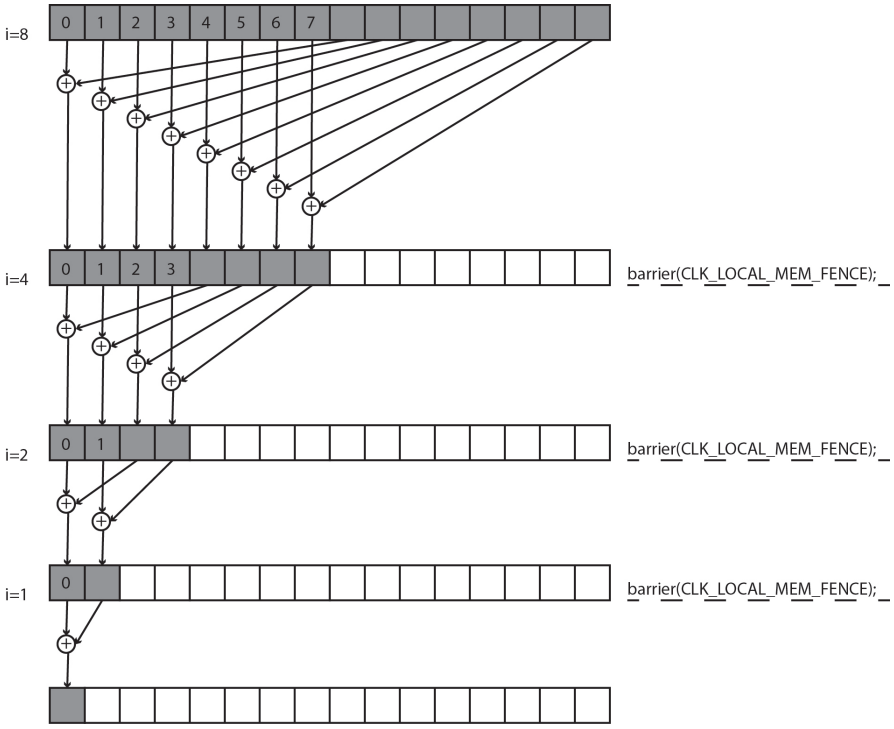


Fig. 5.12: Summation reduction.

After we have completed one step, we have the same restriction we did after computing all the pairwise products. Before we can read the values we just stored in `ProductsWG`, we need to ensure that every thread that needs to write to `ProductsWG` has already done so. The `barrier(CLK_LOCAL_MEM_FENCE)` after the assignment ensures this condition is met. It is important to note that when using barrier, all work-items in the work-group must execute the barrier function. If the barrier function is called within a conditional statement, it is important to ensure that all work-items in the work-group enter the conditional statement to execute the barrier. For example, the following code is an illegal use of barrier because the barrier will not be encountered by all work-items:

```
if (iLID < i) {
   ProductsWG[iLID] += ProductsWG[iLID+i];
   barrier(CLK_LOCAL_MEM_FENCE);
}
```

Any work-item with the local index `iLID` greater than or equal to `i` will never execute the `barrier(CLK_LOCAL_MEM_FENCE)`. Because of the guarantee that no instruction after a `barrier(CLK_LOCAL_MEM_FENCE)` can be executed before every work-item of the work-group has executed it, the hardware simply continues to wait

Reduction

In computer science, the reduction is a special type of operation that is commonly used in parallel programming to reduce the elements of an array into a single result.

for these work-items. This effectively hangs the processor because it results in the GPU waiting for something that will never happen. Such a kernel will actually cause the GPU to stop responding, forcing you to kill your program.

After termination of the summation reduction, each work-group has a single number remaining. This number is sitting in the first entry of the `ProductsWG` buffer and is the sum of every pairwise product the work-items in that work-group computed. We now store this single value to global memory and end our kernel:

```
if (iLID == 0) {
   c[iWGID] = ProductsWG[0];
}
```

As there is only one element from `ProductsWG` that is transferred to global memory, only a single thread needs to perform this operation. Since each work-group writes exactly one value to the global array `c`, we can simply index it by `WGID`, which is the work-group index.

We are left with an array `c`, each entry of which contains the sum produced by one of the parallel work-groups. The last step of the dot product is to sum the entries of `c`. Because array `c` is relatively small, we return control to the host and let the CPU finish the final step of the addition, summing the array `c`.

Listing 5.21 shows the entire kernel function for dot product using shared memory and summation reduction.

```
1  //*************************************************
2  // OpenCL Kernel Function for dot product
3  // using shared memory snd summation reduction
4  __kernel void DotProductShared(__global float* a,
5                                 __global float* b,
6                                 __global float* c,
7                                 __local* ProductsWG,
8                                 int iNumElements)
9  {
10
11     // work-item global index
12     int iGID = get_global_id(0);
13     // work-item local index
14     int iLID = get_local_id(0);
15     // work-group index
16     int iWGID = get_group_id(0);
17     // how many work-items are in WG?
18     int iWGS = get_local_size(0);
19
20     float temp = 0.0;
21     while (iGID < iNumElements) {
22         // multiply adjacent elements
23         temp += a[iGID] * b[iGID];
24         iGID += get_global_size(0);
```

```
    }
    //store the product
    ProductsWG[iLID] = temp;

    // wait for all threads in WG:
    barrier(CLK_LOCAL_MEM_FENCE);

    // Summation reduction:
    int i = iWGS/2;
    while(i!=0){
        if (iLID < i) {
            ProductsWG[iLID] += ProductsWG[iLID+i];
        }
        barrier(CLK_LOCAL_MEM_FENCE);
        i=i/2;
    }

    // store partial dot product into global memory:
    if (iLID == 0) {
        c[iWGID] = ProductsWG[0];
    }
}
```

Listing 5.21: Vector Dot Product Kernel - implementation using local memory and summation reduction

In the host code for this example we should create the `bufferC` memory object that will hold `szGlobalWorkSize/szLocalWorkSize` partial dot products. Listing 5.22 shows how to create `bufferC`.

```

size_t datasize_c = sizeof(cl_float) * (szGlobalWorkSize/↩
    szLocalWorkSize);

// Use clCreateBuffer() to create a buffer object (d_C)
// with enough space to hold the output data
bufferC = clCreateBuffer(
                         context,
                         CL_MEM_READ_WRITE,
                         datasize_c,
                         NULL,
                         &status);
```

Listing 5.22: Create `bufferC` for dot product using local memory

Listing 5.23 shows how to set kernel arguments.

```
//*****************************************************
// STEP 9: Set the kernel arguments
//*****************************************************
// Set the Argument values
ciErr = clSetKernelArg(ckKernel,
                       0,
                       sizeof(cl_mem),
                       (void*)&bufferA);
ciErr |= clSetKernelArg(ckKernel,
                        1,
                        sizeof(cl_mem),
                        (void*)&bufferB);
ciErr |= clSetKernelArg(ckKernel,
                        2,
```

```
15                          sizeof(cl_mem),
16                          (void*)&bufferC);
17 ciErr |= clSetKernelArg(ckKernel,
18                          3,
19                          sizeof(float) * szLocalWorkSize,
20                          NULL);
21 ciErr |= clSetKernelArg(ckKernel,
22                          4,
23                          sizeof(cl_int),
24                          (void*)&iNumElements);
```

Listing 5.23: Set kernel arguments for dot product using shared memory

The argument with index 3 is used to create local memory buffer of size `sizeof(float) * szLocalWorkSize` for each work-group. As this argument is declared in the kernel function with the `__local` qualifier, the last entry to `clSetKernelArg` must be NULL.

Results for dot product of two vectors of size 67108864 ($512 \times 512 \times 256$) on an Apple laptop with an Intel GPU are:

```
Kernel execution time = 0.503470 s
Result = 33554432.000000
```

5.3.5 Naive matrix multiplication in OpenCL

This section describes a matrix multiplication application using OpenCL for GPUs in a step-by-step approach. We will start with the most basic version (naive) where focus will be on the code structure for the host application and the OpenCL GPU kernels. The naive implementation is rather straightforward, but it gives us a nice starting point for further optimization. For simplicity of presentation, we will consider only square matrices whose dimensions are integral multiples of 32 on a side. Matrix multiplication is a key building block for dense linear algebra and the same pattern of computation is used in many other algorithms. We will start with simple version first to illustrate basic features of memory and work-item management in OpenCL programs. After that we will extend to version which employs local memory.

Before starting, it is helpful to briefly recap how a matrix-matrix multiplication is computed. The element $c_{i,j}$ of **C** is the dot product of the i-th row of **A** and the j-th column of **B**. The matrix multiplication of two square matrices is illustrated in Figure 5.13. For example, as illustrated in Figure 5.13, the element $c_{5,2}$ is the dot product of the row 5 of **A** and the column 2 of **B**.

To implement matrix multiplication of two square matrices of dimension $N \times N$, we will launch $N \times N$ work-items. Indexing of work-items in NDRange will correspond to 2D indexing of the matrices. Work-item (i, j) will calculate the element $c_{i,j}$ using row i of **A** and column j of **B**. So, each work-item loads one row of matrix **A** and one column of matrix **B** from global memory, do the dot product, and store the result back to matrix **C** in the global memory. The matrix **A** is therefore read N

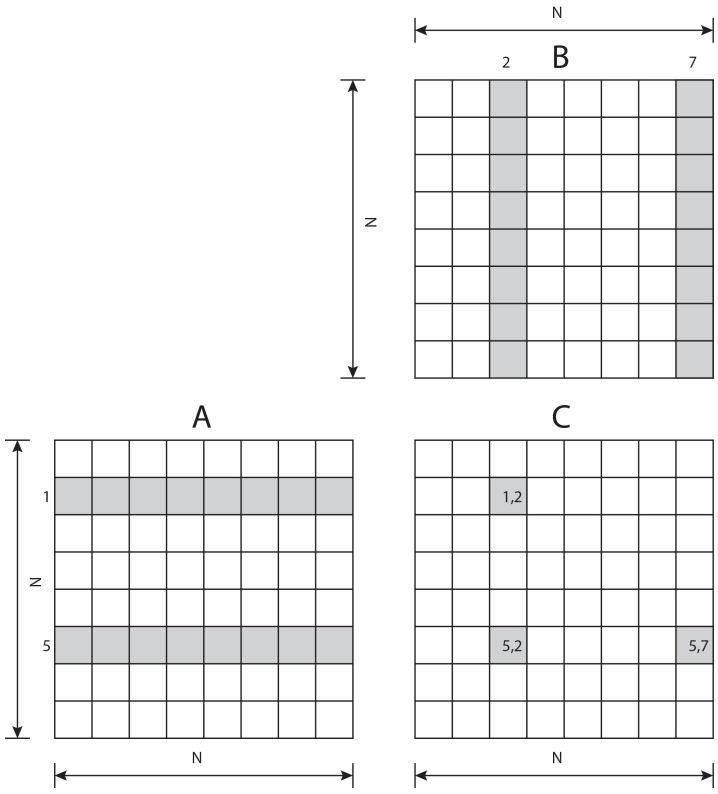


Fig. 5.13: Matrix multiplication.

times from global memory and **B** is read N times from global memory. The simple version of matrix multiplication can be implemented in the plain C language using 3 nested loops as in Listing 5.24. We assume data to be stored in row-major order (C-style).

```
void matrixmul(float *matrixA,
               float *matrixB,
               float *matrixC,
               int N) {

    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            for (int k = 0; k < N; k++) {
               matrixC[i*N + j] +=
                  matrixA[i * N + k] * matrixB[k * N + j];
            }
        }
    }
}
```

Listing 5.24: Simple matrix multiplication in C

Let's now implement the simple matrix multiplication in OpenCL.

The naive multiplication kernel

To implement the above matrix multiplication in OpenCL $N \times N$ work-items will be needed. Let's have each work-item compute an element of the matrix **C**. Each work-item should first discover its ID within 2D NDRange and compute the corresponding element of **C**. Listing 5.25 shows the kernel function for naive matrix multiplication.

```
// OpenCL Kernel Function for naive matrix multiplication
__kernel void matrixmulNaive(
                            __global float* matrixA,
                            __global float* matrixB,
                            __global float* matrixC,
                            int N) {

    // global thread index
    int xGID = get_global_id(0); // column in NDRange
    int yGID = get_global_id(1); // row in NDRange

    float dotprod = 0.0;

    // each work item calculates one element of the matrix C:
    for (int i = 0; i < N; i++) {
        dotprod += matrixA[yGID * N + i] * matrixB[i * N + xGID];
    }
    matrixC[yGID * N + xGID] = dotprod;
}
```

Listing 5.25: The naive multiplication kernel

Each work item first discovers its global ID in 2D NDRange. The index of the column `xGID` is obtained from the first dimension of NDRange using `get_global_id(0)`. Similarly, the index of the row `yGID` is obtained from the second dimension of NDRange using `get_global_id(1)`. After obtaining its glogal ID, each work-item do the dot product between the `yGID`-th row of **A** and the `xGID`-th column of **B**. The dot product is stored to the element in the `yGID`-th row and the `xGID`-th column of **C**.

The host code

As we learned previously, the host code should probe for devices, create context, create buffers and compile the OpenCL program containing kernels. These steps are the same as in the vector addition example from Subsection 5.3.1. Assuming you have already initialized OpenCL, created the context and the qwueue, and created the appropriate buffers and memory copies. Listing 5.26 shows how to compile the kernel for naive matrix multiplication.

```
//*************************************************
// STEP 8: Create and compile the kernel
```

```
//*****************************************************
ckKernel = clCreateKernel(
                          cpProgram,
                          "matrixmulNaive",
                          &ciErr);
if (!ckKernel || ciErr != CL_SUCCESS)
{
    printf("Error: Failed to create compute kernel!\n");
    exit(1);
}
```

Listing 5.26: Create and compile the kernel for naive matrix multiplication

Prior to launch the kernel we should set the kernel arguments as in Listing 5.27.

```
//*****************************************************
// STEP 9: Set the kernel arguments
//*****************************************************
ciErr = clSetKernelArg(ckKernel, 0, sizeof(cl_mem), (void*)&bufferA);
ciErr |= clSetKernelArg(ckKernel, 1, sizeof(cl_mem), (void*)&bufferB);
ciErr |= clSetKernelArg(ckKernel, 2, sizeof(cl_mem), (void*)&bufferC);
ciErr |= clSetKernelArg(ckKernel, 3, sizeof(cl_int), (void*)&iRows);
```

Listing 5.27: Set the kernel arguments for naive matrix multiplication

Finally, we are ready to launch the kernel `matrixmulNaive`. Listing 5.28 shows how you launch the kernel.

```
// ******************************************************
// Start Core sequence... copy input data to GPU, compute,
//     copy results back

// set and log Global and Local work size dimensions
const cl_int iWI = 16;
const size_t szLocalWorkSize[2] = { iWI, iWI };
const size_t szGlobalWorkSize[2] = { iRows, iRows };
cl_event kernelevent;

//*****************************************************
// STEP 10: Enqueue the kernel for execution
//*****************************************************
ciErr = clEnqueueNDRangeKernel(
                               cmdQueue,
                               ckKernel,
                               2,
                               NULL,
                               szGlobalWorkSize,
                               szLocalWorkSize,
                               0,
                               NULL,
                               &kernelevent);
```

Listing 5.28: Enqueue the kernel for naive matrix multiplication

As can be seen from the code in Listing 5.28, NDRange is of 2D size (`iRows`, `iRows`). That means that we launch (`iRows`, `iRows`) work-items. These work-items are further grouped into work-groups of dimension (16,16). If for example the size of matrices (`iRows`, `iRows`) is (4096, 4096), we launch 256×256 work-groups. As the number of work-groups is probably larger than the number of compute-units

present in a GPU, we keep all compute-units busy. Recall that we should always keep the occupancy high, because this is a way to hide latency when executing instructions.

Execution time for naive matrix multiplication of two square matrices of size 3584×3584 on an Apple laptop with an Intel GPU is:

```
Kernel execution time = 30.154823 s
```

5.3.6 Tiled matrix multiplication in OpenCL

Looking at the loop in the kernel code from Listing 5.25, we can notice that each work-item loads $2 \times N$ elements from global memory - two for each iteration through the loop, one from the matrix **A** and one from the matrix **B**. Since accesses to global memory are relatively slow, this can slow down the kernel, leaving the work-items idle for hundreds of clock cycles, for each access. Also we can notice that for each element of **C** in a row we use the same row of **A** and that each work-item in a work-group uses the same columns of **B**.

But not only are we accessing the GPU's off-chip memory way too much, we don't even care about memory coalescing! Assuming row-major order when storing matrices in global memory, the elements from the matrix **A** are accessed with unit stride, while elements from the matrix **B** are accessed with stride N.

Recall from Subsection 5.1.4 that, to ensure memory coalescing, we want work-items from the same warp to access contiguous elements in memory so to minimize the number of required memory transactions. As work-items of the same warp access 32 contiguous floating-point elements from the same row of **A**, all these elements fall into the same 128-bytes segment and data is delivered in a single transaction. On the other hand, work-items of the same warp access 32 floating-point elements from **B** that are 4N bytes apart, so for each element from the matrix **B** a new memory transaction is needed. Although the GPU's caches probably will help us out a bit, we can get much more performance by *manually caching sub-blocks of the matrices (tiles)* in the GPU's on-chip local memory.

In other words, one way to reduce the number of accesses to global memory is to have the work-items load portions of matrices **A** and **B** into local memory, where we can access them much more quickly. So we will use local memory to avoid non-coalesced global memory access. Ideally, we would load both matrices entirely into local memory, but unfortunately, local memory is a rather limited resource and cannot hold two large matrices. Recall that older devices have 16kB of local memory per compute unit, and more recent devices have 48kB of local memory per compute unit. So we will content ourselves with loading portions of **A** and **B** into local memory as needed, and making as much use of them as possible while they are there.

Assume that we multiply two matrices as shown in Figure 5.14. To calculate the elements of the square submatrix **C** (*tile* C), we should multiply the corresponding

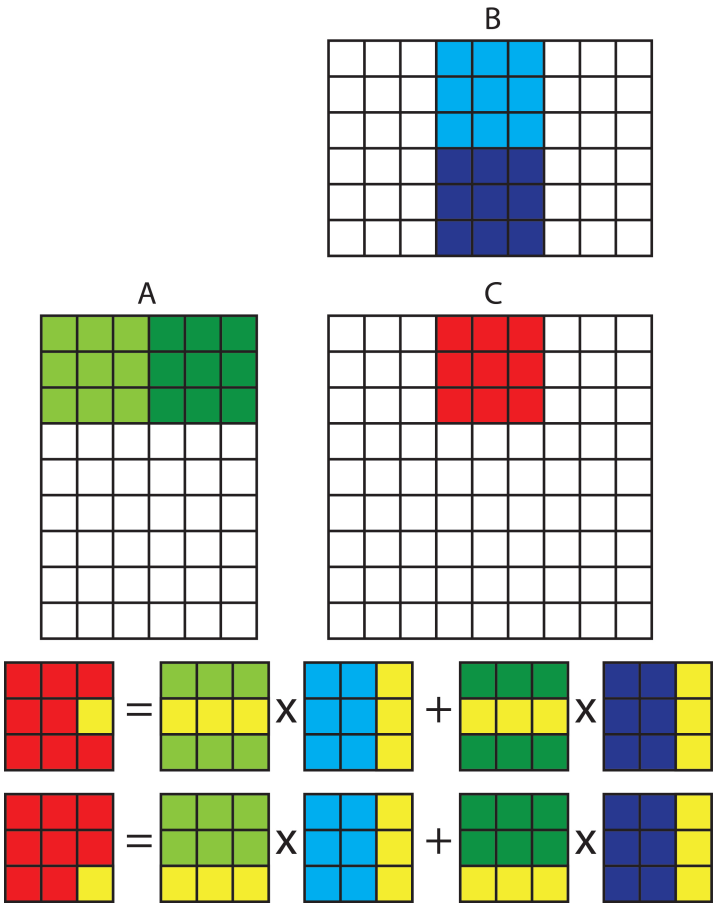


Fig. 5.14: Matrix multiplication using tiles.

rows and columns of matrices **A** and **B**. Also, we can subdivide the matrices **A** and **B** into submatrices (*tiles*) such as shown in Figure 5.14. Now we can multiply the corresponding row and the column from the **A** and **B** tiles and sum up these partial products. The process is shown in the lower part of Figure 5.14. We can also observe that the individual rows and columns in tiles **A** and **B** are accessed several times. For example, in the 3x3 tiles from Figure 5.14, all elements on the same row of the tile **C** are computed using the same data of the **A** tiles and all elements on the same column of the submatrix **C** are computed using the same data of the **B** tile. As the tiles are in local memory, these accesses are fast.

The idea of using tiles in matrix multiplication is as follows. The number of work-items that we start is equal to the number of elements in the matrix. Each work-item will be responsible for computing one element of the product matrix **C**.

The index of the element in the matrix is equal to the global index of a work-item in NDrange. At the same time, we create the same number of work-groups as is the number of tiles. The number of elements in a tile will be equal to the number of threads in a work-group. This means that the element index within a tile will be the same as the local index in the group.

For reference, consider the matrix multiplication in Figure 5.15. All matrices are of size 8×8, so we will have 64 work-items in NDrange. We divide matrices **A**, **B** and **C** in non-overlapping sub-blocks (tiles) of size $TW \times TW$, where $TW = 4$ as in Figure 5.15. Let's also suppose that tiles are indexed starting in the upper left corner. Now, consider the element $c_{5,2}$ in the matrix **C**, in Figure 5.15. The element $c_{5,2}$ falls into the tile (0,1). The work-item responsible for computing the element $c_{5,2}$ has the global row index 5 and the global column index 2. Also the same work-item has the local row index 1 and local column index 2. This work-item computes the element $c_{5,2}$ in **C** by multiplying together row 5 in **A**, and column 2 in **B**, but it will do it in pieces using tiles. As we already said, all work-items responsible for computing

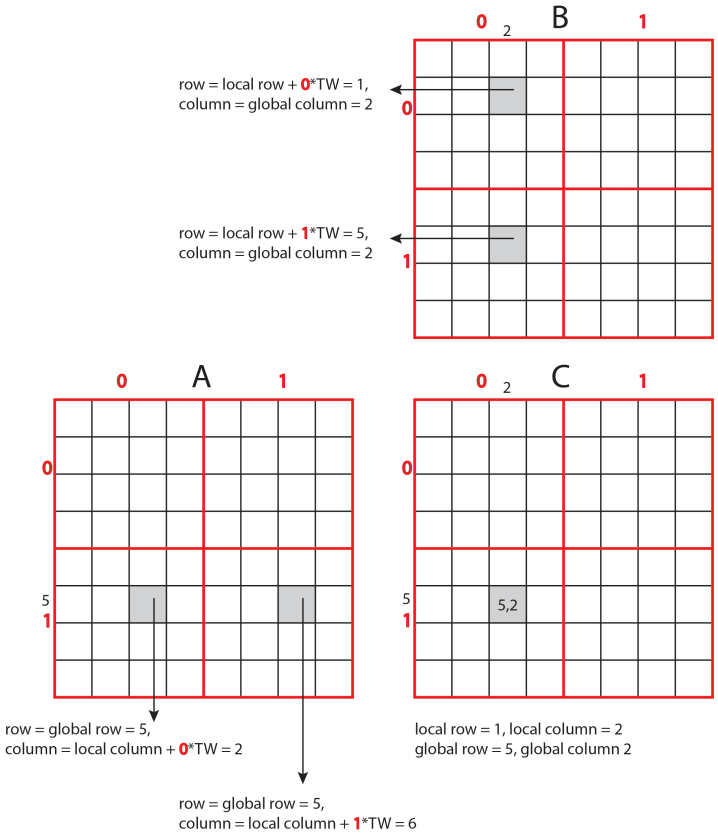


Fig. 5.15: Matrix multiplication with tiles.

elements of the same tile in the matrix **C** should be in the same work-group. Let's explain this process for the work-item that computes the element $c_{5,2}$. The work-item should access the tiles (0,1) and (1,1) from **A** and tiles (0,0) and (1,1) from **B**. The computation is performed in two steps. First, the work-item computes dot product between the row 1 from the tile (0,1) in **A** and the column 2 from the tile (0,0) in **B**. In the second step, the same work-item computes dot product between the the row 1 from the tile (1,1) in **A** and the column 2 from the tile (1,0) in **B**. Finally it adds this dot product to the one computed in the first step.

If we want to compute the first dot product as fast as possible, the elements form the row 1 from the tile (0,1) in **A** and the elements from the column 2 from the tile (0,0) in **B** should be in the local memory. The same is true for all rows in the tile tile (0,1) in **A** and all columns in the tile (0,0) in **B**, because all work-items from the same work-group will access this elements concurrently. Also, ones the first step is finished, the same will be true for the second step but this time for the tile (1,1) in **A** and the tile (1,0) in **B**.

So, before every step, all work-items from the same work-group should perform a *collaborative load* of tiles A and B into local memory. This is performed in such a way that the work-item in the $i-th$ local row and the $j-th$ local column performs two loads from global memory per tile: the element with local index (i, j) from the corresponding tile in matrix **A** and the element with local index (i, j) from the corresponding tile in matrix **B**. Figure 5.15 illustrates this process. For example, the work item that computes the element $c_{5,2}$ reads:

1. the elements $a_{5,2}$ and $b_{1,2}$ in the first step, and
2. the elements $a_{5,6}$ and $b_{5,2}$ in the second step.

Where is the benefit of using tiles? If we load the left-most (0,1) tile of matrix **A** into local memory, and the top-most (0,0) of those tiles of matrix **B** into local memory, then we can compute the first $TW \times TW$ products and add them together just by reading from local memory. But here is the benefit: as long as we have those tiles in local memory, every work-item from the work-group computing a tile form **C** can compute that portion of their sum from the same data in local memory. When each work item has computed this sum, we can load the next $TW \times TW$ tiles from **A** and **B**, and continue adding the term-by-term products to our value in **C**. And after all of the tiles have been processed, we will have computed our entries in **C**.

The tiled multiplication kernel

The kernel code for the tiled matrix multiplication is shown in Listing 5.29.

```
#define TILE_WIDTH 16

// OpenCL Kernel Function for tiled matrix multiplication
__kernel void matrixmulTiled(
                              __global float* matrixA,
                              __global float* matrixB,
                              __global float* matrixC,
```

```
                              int N) {


    // Local memory to fit the tiles
    __local float matrixAsub[TILE_WIDTH][TILE_WIDTH];
    __local float matrixBsub[TILE_WIDTH][TILE_WIDTH];

    // global thread index
    int xGID = get_global_id(0); // column in NDRange
    int yGID = get_global_id(1); // row in NDRange

    // local thread index
    int xLID = get_local_id(0); // column in tile
    int yLID = get_local_id(1); // row in tile

    float dotprod = 0.0;

    for(int tile = 0; tile < N/TILE_WIDTH; tile++){
        // Collaborative loading of tiles into shared memory:
        // Load a tile of matrixA into local memory
        matrixAsub[yLID][xLID] =
           matrixA[yGID * N + (xLID + tile*TILE_WIDTH)];
        // Load a tile of matrixB into local memory
        matrixBsub[yLID][xLID] =
           matrixB[(yLID + tile*TILE_WIDTH) * N  + xGID];

        // Synchronise to make sure the tiles are loaded
        barrier(CLK_LOCAL_MEM_FENCE);

        for (int i = 0; i < TILE_WIDTH; i++) {
            dotprod +=
               matrixAsub[yLID][i] * matrixBsub[i][xLID];
        }

        // Wait for other work-items to finish
        //      before loading next tile
        barrier(CLK_LOCAL_MEM_FENCE);
    }

    matrixC[yGID * N + xGID] = dotprod;
}
```

Listing 5.29: The tiled multiplication kernel

Tiles are stored in `matrixAsub` and `matrixBsub`. Each work-item finds its global index and its local index. The outer loop goes through all the tiles necessary to calculate the products in **C**. Each work-item in the work-group in one iteration of the outer loop first reads its elements from the global memory and writes them to the tile element with its local index. After loading its elements, each work-item waits at the barrier until the tiles are loaded. Then, in the innermost loop, each work-item calculates dot product between a row `yLID` form the tile `matrixAsub` and the column `xLID` from the tile `matrixBsub`. After that the work-item waits again at the barrier for the other work-items to finish their dot products. Then all work-items load next tiles and repeat the process.

The host code

To implement tiling, we'll leave our host code from the previous naive kernel intact. The only thing we should change is to create and compile the appropriate kernel function:

```
//*****************************************************
// STEP 8: Create and compile the kernel
//*****************************************************
ckKernel = clCreateKernel(
                           cpProgram,
                           "matrixmulTiled",
                           &ciErr);
if (!ckKernel || ciErr != CL_SUCCESS)
{
    printf("Error: Failed to create compute kernel!\n");
    exit(1);
}
```

Listing 5.30: Create and compile the kernel for tiled matrix multiplication

Note that it already uses 2D work-groups of 16 by 16. This means that the tiles are also 16 by 16.

Execution time for tiled matrix multiplication of two square matrices of size 3584×3584 on an Apple laptop with an Intel GPU is:

```
Kernel execution time = 16.409384 s
```

5.4 Exercises

1. To verify that you understand how to control the argument definitions for a kernel, modify the kernel in Listing 5.3 so it adds four vectors together. Modify the host code to define four vectors and associate them with relevant kernel arguments. Read back the final result and verify that it is correct.
2. Use local memory to minimize memory movement costs and optimize performance of the matrix multiplication kernel in Listing 5.25. Modify the kernel so that each work-item copies its own row of **A** into local memory. Report kernel execution time.
3. Modify the kernel from the previous exercise so that each work-group collaboratively copies its own column of **B** into local memory. Report kernel execution time.
4. Write an OpenCL program that computes the Mandelbrot set. Start with the program in Listing 3.21.
5. Write an OpenCL program that computes π. Start with the program in Listing 3.15. Hint: the paralellization is similar to the parallelization of a dot product.
6. Write an OpenCL program that transposes a matrix. Use local memory and collaboratively reads to minimize memory movement costs and optimize performance of the kernel.

7. Given an input array $\{a_0, a_1, , a_{n-1}\}$ in pointer `d_a`, write an OpenCL program that stores the reversed array $\{a_{n-1}, a_{n-2}, , a_0\}$ in pointer `d_b`. Use multiple blocks. Trya to revert data in local memory. Hint: using work-groups and local memory revert data in array slices. Than revert slices in global memory.
8. Write an OpenCL program to detect edges on black and with images using the Sobel filter.

5.5 Bibliographical notes

The primary source of information including all details of OpenCL is available at Khronos web site [15] where the complete reference guide is available. Another good online source of OpenCL tutorials and dozen of examples is *HandsOnOpenCL training course* [14]. A comprehensive hands-on presentation of OpenCL can be found in the book *OpenCL in Action* by Matthew Scarpino [24]. A gentle introduction to OpenCL by the same author can be found in [23]. The books by Munshi et al. [17] and Gaster et al. [11] provide a deep dive into OpenCL.

Part III
Engineering

The aim of Part III is to explain why a parallel program can be more or less efficient. A basic approaches are described for the performance evaluation and analysis of parallel programs. Instead of analyzing complex applications, we focus on two simple cases, i.e. a parallel **computation of number** π, by using numerical integration, and a solution of simplified **partial differential equation** on 1-D domain, by using explicit solution methodology. Both cases, already mentioned in previous chapters, even so simple, they already incorporate most of possible pitfalls that could arise during their parallelization. The first case, computation of *pi*, requires just a few communication among parallel tasks, while in the explicit solution of PDE, each process communicate with its neighbors in every time step.

Besides these two cases, we also evaluate the **Seam Carving** algorithm in terms of performance on CPU and a GPU platform. Seam Carving is an image processing algorithm in 2-D domain and as such appropriate for implementation on GPU platforms. It comprises a few steps of which some cannot be effectively parallelized.

Parallel programs run on adequate platforms, i.e. multi-core computers, interconnected computers or computing clusters, and GPU accelerators. After an implementation of any parallel program, several questions remain to be answered, e.g.:

- How the execution time decreases with larger number of processors?
- How many processors are optimal for a specific task?
- Will execution time always decrease, if the number of processors is increased?
- Which parallelization methodology provides the best results?
- and similar.

We will answer the above questions by running the programs with different parameters, e.g. size of the computation domain and the number of processors. We will follow also the execution efficiency and limitations that are specific for each of the three parallel methodologies: OpenMP, MPI, and OpenCL.

An electronic extension of the Engineering part will be permanently available on a book web, hosted by Springer server. Our aim is that it become a vivid forum of readers, students, teachers and other developers. We expect your inputs in a form of your own cases, solutions, comments, and proposals. Soon after the publication of this book more complex cases will be provided, i.e. a numerical solution of a 2-D diffusion equation, a simulation of N-body interactions with possible application in molecular dynamics, and similar.

Each of engineering cases will be introduced with a basic description of the selected problem, sequential algorithm and its solution methodology. Then, for all considered parallelization approaches: OpenMP, MPI and OpenCL, initial parallel algorithms will be developed and their expected performance will be estimated. Results will be compared in terms of programming complexity, execution time, and scalability. The complete implementations will be provided with an adequate program code. Any improvements and feed back from all users are welcome.

Chapter 6
Engineering: Parallel computation of the number π

A detailed description of the parallel computation of π is available in Chapter 3 Example 3.4 and in Chapter 4 Example 4.4. The solution methodology relies on a numerical integration of unit circle:

$$\pi = 4\int_0^1 \sqrt{1-x^2}dx$$

that is in a direct relation with the value of π. The numerical integration is performed by calculation and summation of all N sub-interval areas. A sequential version of the algorithm in a pseudo-code, which results in an approximate value of π, is given below:

Algorithm 1 SEQUENTIAL ALGORITHM: COMPUTE_π

Input: N - number of sub-intervals on interval [0, 1]

1: **for** $i = 1 \ldots N$ **do**
2: $\quad x_i = (1/N)(i-0.5)$
3: $\quad y_i = \sqrt{1-x_i^2}$
4: $\quad Pi = Pi + 4(y_i/N)$
5: **end for**

Output: Pi - an approximation for the number π

We validate the Algorithm 1 on a single computer in order to prove that its correct behaviour. It is expected that with an increased number of sub-intervals N, the approximation of π will become better and better, which should be confirmed by calculated absolute error of approximate π value. This is easy, because we know the π value with arbitrary accuracy. However, with the increased N the run-time will also increase. Embedding the existing MPI program from Listing 4.5 in an additional `for` loop that increases the number of intervals by a factor of two, followed by compiling and running the program:

```
>mpiexec -n 1 MSMPIPi
```

on a HP EliteBook 840 notebook, based on Intel Core 64 bit processor i7-7500U CPU with 2 physical cores and 4 logical processor, on MS Windows 10 operating system with Visual Studio 2017 compiler, we get the results shown in Fig. 6.1. Note, that for all presented MPI experiments in this book, the same notebook was used. We compile in `Release` mode with optimization for maximal speed, e.g. `/O2`.

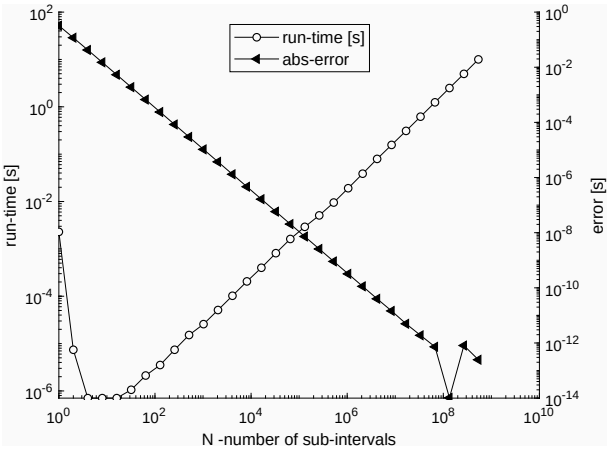


Fig. 6.1: Run-time and absolute error on a single MPI process in computation of π as a function of number of sub-intervals N.

To see the full response, the results are shown in logarithmic scale on both axes. The run-time mostly increases as expected, except with a few smallest values of sub-intervals, where the impact of MPI program set-up time, cache memory or interactions with operating system could be present. In the same way, the approximation error becomes smaller and smaller, until the largest number of intervals, where a small jump is presents, possibly because of a limited precision of the floating point arithmetic.

The next step is to find out the most efficient way to parallelize the problem, i.e. to engage a greater number of cooperating processors in order to speed-up the program execution. Even that the sequential Algorithm 1 is very simple it implies most of the problems that arise also in more complex examples. First, the program needs to distribute tasks among cooperating processors in a balanced way. A relatively small portion of data should be communicated to cooperating processes, because the processes will generate their local data by a common equation for a unit circle. All processes have to implement their local computation of partial sums, and finally, the partial results should be assembled, usually by a global communication, in a host process to be available for users.

Regarding sequential Algorithm 1 algorithm, we see that the calculation of each sub-interval area is independent, consequently, the algorithm has a potential to be

parallelized. In order to make the calculation parallel, we will use domain decomposition approach and master-slave implementation. Because all values of y_i can be calculated locally and because the domain decomposition is known explicitly, there is no need for a massive data transfer between the master process and slave processes. The master process will just broadcast the number of intervals. Then the local integration will run in parallel on all processes. Finally, the master process reduce the partial sums into the final value of π. The parallelized algorithm is shown below:

Algorithm 2 PARALLEL ALGORITHM: COMPUTE_π

Input: N - number of sub-intervals on interval [0, 1]

1: Get *myID* and the number of cooperating processes p
2: Master broadcast N to all processes
3: Compute a shorter **for** loop:
4: **for** $j = 1 \ldots N/p$ **do**
5: $\quad x_j = (1/N)(j - 0.5)$
6: $\quad y_j = \sqrt{1 - x_j^2}$
7: $\quad Pj = Pj + 4(y_j/N)$
8: **end for**
9: Master reduce partial sums Pj to the final result Pi

Output: Pi - approximate value of π

We have learned from this simple example that, beside the calculation, there are other tasks to be done (i) domain decomposition (ii) their distribution and (iii) assembling of the final result, which are inherently sequential, and therefore limit the final speed-up. We further see that all processes are not identical. Some of the processes are slaves because they just calculate their portion of data. The master process has to distribute number of intervals and to gather and sum-up the local results. Parallel implementation approaches on different computing platforms differs significantly, therefore their results are presented in the following sections, separately for OpenMP, MPI, and OpenCL.

6.1 OpenMP

Computing π on a multicore processor has been covered in Chapter 3.

The numerical integration of a unit square (the part of it that lies in the 1st quadrant) has been explained in Example 3.4 where the program for computing π is shown Listing 3.15. To analyze the performance of the program it has been run on a quadcore processor with hyperthreading (Intel Core i7 6700HQ). For 10^9 subintervals of the interval $[0,1]$ (when the error is approximately 10^{-8}), the results are shown in Figure 6.2: the bars show the measured wall clock time and the dashed

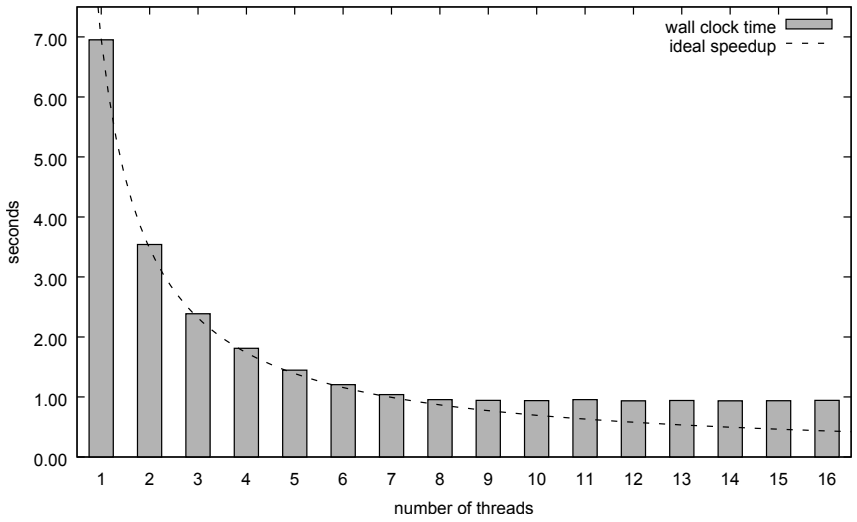


Fig. 6.2: Computing π using the numerical integration of the unit circle using 10^9 intervals on a quadcore processor with hyperthreading.

curve illustrates the expected wall clock time in case of the ideal speedup in regard to the number of threads used.

The wall clock time decreases when the number of threads increases, but only up to the number of logical cores the processor can provide. Once the number of threads exceeds the number of logical cores, the program (its OpenMP runtime component, to be precise) places multiple threads on the same core and no reduction of wall clock time can be gained.

In fact, one must observe that up to the number of logical cores, almost ideal speedup is achieved. This is not to be expected very often. In this case, however, it is a consequence of the fact the the entire computation is almost perfectly parallelizable, with the exception of the final reduction. But if 10^9 intervals are divided among 8 threads, the time of the reduction becomes insignificant if compared to the computation of the local sums.

As shown in Example 3.5, π can also be computed by random shooting into the square $[0,1] \times [0,1]$ and count the number of shots that hit inside the unit square. The program for computing π using this method is shown in Listing 3.18. As with the numberical integration, the program has been tested on a quadcore processor with hyperthreading (Intel Core i7 6700HQ). For 10^8 shots, the measured wall clock time is shown in Figure 6.3. Again, the dashed line illustrates the expected wall clock time in case of the ideal speedup in regard to the number of threads used.

As can be seen in Figure 6.3, (almost) ideal speedup is achieved only for up to 4 threads, i.e., for one thread per physical, not logical core. That implies that instructions and memory accesses of threads placed on the same physical core result

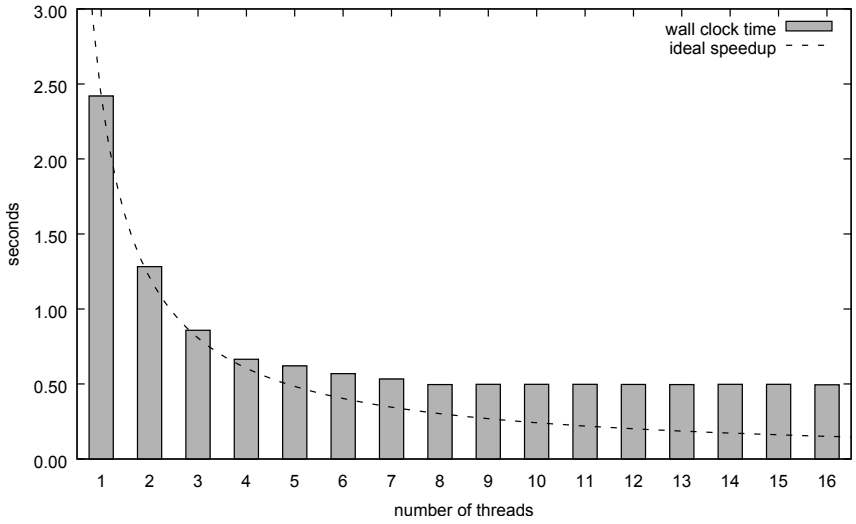


Fig. 6.3: Computing π using random shooting into the square $[0,1] \times [0,1]$ using 10^8 shots.

in too many conflicts to sustain the speedup and trully benefit from multithreading. This can happen and it is a lesson not to be forgotten.

Even though the wall clock time is what it matters in the end, the CPU time has been measured as well. In Figure 6.4 the total amount of CPU time needed for computing π using both methods explained in Chapter 3, namely numerical integration and random shooting, is shown.

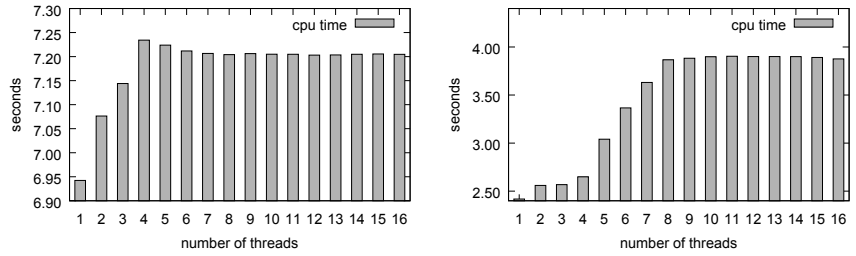


Fig. 6.4: The total CPU time needed to compute π using numerical integration (left) and random shooting (right).

Although the wall clock time shown in Figures 6.2 in 6.3 decreases with the number of threads the total amount of CPU time increases. These can be expected, since more threads requires more administrative tasks from the OpenMP runtime.

6.2 MPI

An MPI C code for the parallel computation of π value, together with some explanation and comments, are provided in Chapter 4 Listing 4.5. We would like to test the behaviour of run-time as a function of the number of MPI processes p. On the test notebook computer, two cores are present. Taking into account that four logical processors are available, we could expect some speed-up of the execution with up-to four MPI processes. With more than four processes, the run-time could start increasing, because of an MPI overhead. We will test our program with up-to eight processes. Starting a same program on different number of processes can be accomplished by consecutive `mpiexec` commands with appropriate value of parameter `-n` or by a simple bash file that prepares the execution parameters, which are passed to the `main`program through its `argc` and `argv` arguments.

The behaviour of approximation error should be the same as in the case of a single process. In the computation of π, the following number of sub-intervals have been used $N = [5e9, 5e8, 5e7, 5e6]$. Note, that such big numbers of sub-intervals were used because we want to have a computationally complex task, even that the computation of sub-interval areas is quite simple. Usually, in realistic tasks, there is much more computation by itself and tasks become complex automatically. Two smaller values of N have been used to test the impact of the ratio calculation/communication complexity on the program execution. The obtained results for parallel run-times (RT) in seconds and speed-ups (SU), in computation of π, on a notebook computer are shown in Fig. 6.5.

We have first checked that the error in parallel approximation of π is the same as in the case of a single process. The run-time behaves as expected, with the maximum speed-up of 2.6 with four processes and large N. With two processes the speed-up is almost 2, because the physical cores have been allocated. Up to four processes, the speed-up increases but not ideal, because logical processors can not provide the same performance as the physical cores because of hyper-threading technology. The program is actually executed on a shared memory computer with potentially negligible communication delays. However, if N is decreased, e.g. to 5e6 or more, the speed-up is becoming smaller because more processes introduce a larger execution overhead that diminish the speed-up.

Let us finally check the behaviour of the parallel MPI program on a computing cluster. It is built of 36 computing nodes connected in a 6×6 mesh, each with six Gigabit ports to a large Gigabit switch. Computing nodes are built as a dual 64 bit CPU Intel Xeon 5520, each of CPUs with four physical cores (two treads/core) and 6 GB of local memory. The computing cluster runs under server version of Ubuntu

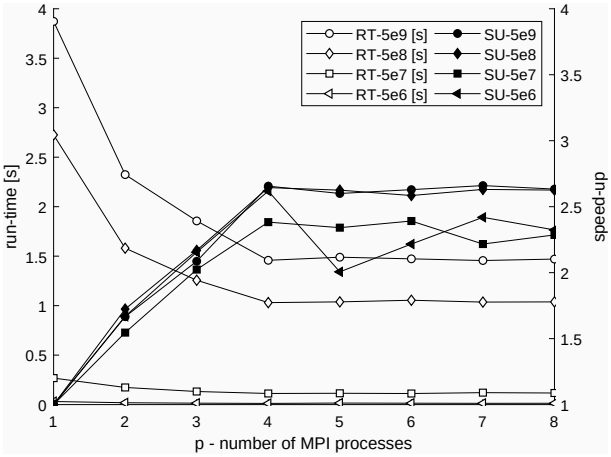


Fig. 6.5: Parallel run-time (RT) and speed-up (SU) in computation of π on a notebook computer for $p = [1,\ldots,8]$ MPI processes and $N = [5e9, 5e8, 5e7, 5e6]$ subintervals.

16.04.3 LTS with GCC Version 7.3 compiler. Only 8 out of 36 interconnected cluster computers (CPUs) have been devoted for our tests, resulting in 32 physical cores. All programs are compiled for maximum speed. Note, that the same computing cluster has been used in all presented MPI tests of this book. The hostfile is:
`k1:4 k2:4 k3:4 k4:4 k5:4 k6:4 k7:4 k8:4`
where `k1...k8` are names of 8 cluster computing nodes.

Because, in the π test case program, there is no significant communication load, and because two treads/core are available, we expect practically ideal speed-up up to 64 MPI processes. Then, if more processes are generated, the speed-up is not so predictable. We will try to explain the results after performing all experiments.

The program is compiled with:

```
>mpicc.mpich -O3 MPI_1DHeat.c -o MPI_1DHeat
```

Parallel program performances are tested on Mpich MPI with various options for `mpiexec.mpich`. First we run `np = 1...128` experiments, for 1 to 128 MPI processes, without default parameters:

```
>mpirun.mpich --hostfile myhosts.mpich.txt -np $np ./MPIPi $N
```

Parameters `N` and `np` are provided from `bash` file as $N = [5e9, 5e8, 5e7, 5e6]$ and $\texttt{np} = [1,\ldots,8]$. The number of MPI processes, i.e. 128, was determined from the command line when running the bash file:

```
>./run.sh 128 > data.txt
```

where `data.txt` is an output file for results. The obtained results for parallel runtimes, in seconds, with default parameters of `mpiexec` (RT-D) and corresponding

speed-ups (SU-D), are shown in Fig. 6.6. For better visibility only two pair of graphs are shown, for largest and smallest N.

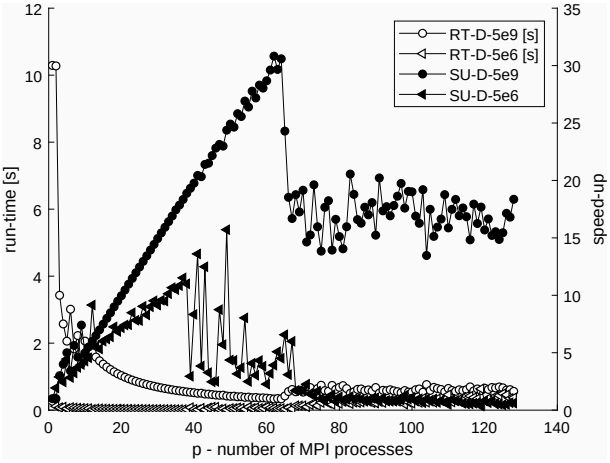


Fig. 6.6: Parallel run-time (RT-D) and speed-up (SU-D) in computations of π on a cluster of 8 interconnected computers with total 32 cores and $p = [1, \ldots, 128]$ MPI processes with default parameters of `mpiexec.mpich`.

Lets look first the speed-up for $N = [5e9$ intervals. We see that the speed-up increases up to 64 processes where reaches its maximal value 32. For more processes, it drops and vandersd around 17. The situation is similar with thousand times smaller number of intervals $N = [5e9$, however, the maximal speed-up is only 5 and for more than 64 processes there is no speed-up. We expected this, because calculation load decreases with smaller number of sub-intervals and the impact of communication and operating system overheads prevail.

We further see that the speed-up scales but not ideal. 64 MPI processes are needed to execute the program 32 times faster as a single computer. The reason could be in the allocation of processes along the physical and logical cores. Therefore we repeat experiments with `mpiexec` parameter `-bind-to core:1`, which forces to run just a single process on each core and possible prevents operating system to move processes around cores. The obtained results for parallel run-times in seconds (RT-B) with processes bound to cores and corresponding speed-ups (SU-B), are shown in Fig. 6.7. The remaining execution parameters are the same as in previous experiment.

The `bind` parameter improves the execution performance with $N = 5e9$ intervals in the sense that the speed-up of 32 is achieved already with 32 processes, which is ideal. But then the speed-up falls abruptly by a factor of 2, possibly because of the fact, that with more than 32 MPI processes, some processing cores must manage two processes, which slows down the whole program execution.

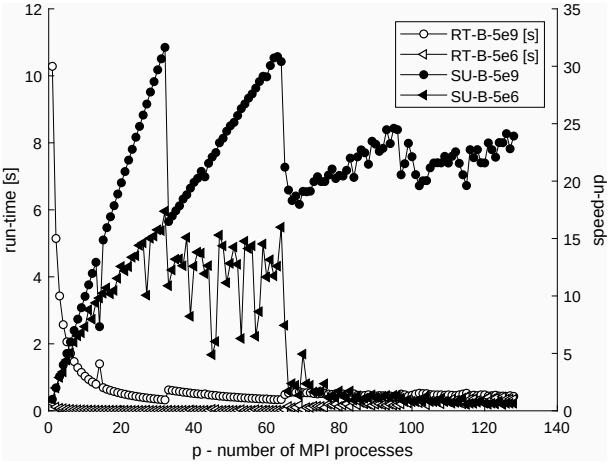


Fig. 6.7: Parallel run-time (RT-B) and speed-up (SU-B) in computations of π on a cluster of 8 interconnected computers for $p = [1, \ldots, 128]$ MPI processes, bound to cores.

We further see that with more than 64 processes speed-ups fall significantly in all tests, which is possible a consequence of inability to use the advantage of hyper-threading. With larger number of processes, larger than the number of cores, on several cores run more than two processes, which slows down the whole program by a factor of 2. Consequently, the slope of speed-up scaling, with more than 32 processes, is also reduced by 2. With this slope, the speed-up reaches the second peak by 64 processes. Then the speed-up falls again to an approximate value of 22.

The speed-up with $N = 5e9$ intervals remains similar as in previous experiment because of low computation load. It is a matter of even more detailed analysis, why the speed-up behaves quite unstable for some cases. The reasons could be in cache memory faults, MPI overheads, collective communication delays, interaction with operating system, etc.

6.3 OpenCL

If we look at Algorithm 1, we can see that it is very similar to dot product calculation covered in Section refsub:dotlocalm. We use the same principle: we will use a buffer in local memory named `LocalPiValues` to store each work-item's running sum of the pi value. This buffer will store `szLocalWorkSize` π values so each work-item in the work-group will have a place to store its temporary result. Then we use the principle of reduction to sum up all π values in the work-group. We now store the final value of each work-group to an array in global memory. Because this array is

relatively small, we return control to the host and let the CPU finish the final step of the addition. Listing 6.1 shows the kernel code for computing π.

```
__kernel void CalculatePiShared(
                            __global float* c,
                            ulong iNumIntervals)
{
    __local float LocalPiValues[256]; // work-group size = 256

    // work-item global index
    int iGID = get_global_id(0);
    // work-item local index
    int iLID = get_local_id(0);
    // work-group index
    int iWGID = get_group_id(0);
    // how many work-items are in WG?
    int iWGS = get_local_size(0);

    float x = 0.0;
    float y = 0.0;
    float pi = 0.0;

    while (iGID < iNumIntervals) {
        x = (float)(1.0f/(float)iNumIntervals)*((float)iGID-0.5f);
        y = (float)sqrt(1.0f - x*x);
        pi += 4.0f * (float)(y/(float)iNumIntervals);
        iGID += get_global_size(0);
    }

    //store the product
    LocalPiValues[iLID] = pi;
    // wait for all threads in WG:
    barrier(CLK_LOCAL_MEM_FENCE);

    // Summation reduction:
    int i = iWGS/2;
    while(i!=0){
        if (iLID < i) {
            LocalPiValues[iLID] += LocalPiValues[iLID+i];
        }
        barrier(CLK_LOCAL_MEM_FENCE);
        i=i/2;
    }

    // store partial dot product into global memory:
    if (iLID == 0) {
        c[iWGID] = LocalPiValues[0];
    }
}
```

Listing 6.1: The compute π kernel.

To analyze the performance of the OpenCL program for computin π, the sequential version has been run on a quadcore processor Intel Core i7 6700HQ running at 2,2 GHz, while the parallel version has been run on an Intel Iris Pro 5200 GPU running at 1,1 GHz. This is a small GPU integrated on the same chip as the CPU and has only 40 processing elements. The results are presented in Table 6.1. We run the kernel in NDrange of size:

```
szLocalWorkSize = 256;      // # of work-items in work-group
szGlobalWorkSize = 256*128; // total # of work-work-items
```

Table 6.1: Experimental results for OpenCL π computation

No. of intervals	CPU time [s]	GPU time [s]	Speedup
10^6	0.01	0.0013	7.69
33×10^6	0.31	0.035	8.86
10^9	9.83	1.07	9.18

As can be seen from the measured execution times, noticeable acceleration is achieved, although we do not achieve the ideal speedup. The main reason for that lies in reduction summation that cannot be fully parallelized. The second reason is the use of complex arithmetic operations (square root). The execution units usually do not have their own unit for such a complex operation, but several execution units share one special-function unit that performs complex operations such as square root, sine, etc.

Chapter 7
Engineering: Parallel solution of 1-D heat equation

Partial differential equations (PDE) are a useful tool for the description of natural phenomena like heat transfer, fluid flow, mechanical stresses, etc. The phenomena are described with spatial and time derivatives. For example, a temperature evolution in a thin isolated bar of length L, with no heat sources, can be described by a PDE of the form:

$$\frac{\partial T(x,t)}{\partial t} = c\frac{\partial^2 T(x,t)}{\partial x^2}$$

where $T(x,t)$ is an unknown temperature at position x in time t, and c is a thermal diffusivity constant with typical values for metals being about 10^{-5} m^2/s. The PDE says that the first derivative of temperature T by t is equal to the second derivative of T by x. To fully determine the solution of the above PDE, initial temperature of the bar, a constant T_0 in our simplified case:

$$T(x,0) = T_0.$$

Finally, fixed temperatures, independent of time, at both ends of the bar are imposed as: $T(0) = T_L$ and $T(L) = T_R$.

In the case of strong solution methodology, the problem domain is discretized in space and the derivatives are approximated by, e.g. finite differences. This results in a system matrix A, three-diagonal in the case of 1-D domain or five-diagonal in 2-D case. To save memory, and because the matrix structure is known, the vectors with old and new temperatures are needed only. The evolution in time can be obtained by an explicit iterative calculation, e.g. Euler method, based on the extrapolation of the current solution and its derivatives into the next time-step, respecting the boundary conditions. A developing solution in time can be obtained by a simple matrix-vector multiplication. If only a stationary solution is desired, then the time derivatives of the solution become zero, and the solution can be obtained in a single step, through a solution of the resulting liner system, $Au = b$, where A is a system matrix, u is a vector of unknown solution values in discretization nodes and b is a vector of boundary conditions.

We will simplify the problem by analyzing 1-D domain only. Note that an extension in 2-D domain, i.e. a plate, is quite straight forward, and can be left for a mini project. For our simplified case, an isolated thin long bar, an exact analytic solution exists. Temperature is spanning in a linear way between the both fixed temperatures at boundaries. However, in real cases, with realistic domain geometry and complex boundary conditions, the analytic solution may not exist. Therefore, an approximate numerical solution is the only option. To find the numerical solution, the domain has to be discretized in space with $j = 1 \ldots N$ points. To simplify the problem the discretization points are equidistant, so $x_{j+1} - x_j = \Delta x$ is a constant. Discretized temperatures $T_j(t)$ for $j = 2 \ldots (N-1)$ solution values in inner points and $T_0 = T_L$ and $T_N = T_R$ are boundary points with fixed boundary conditions.

Finally, we also have to discretize time in equal time-steps $t_{i+1} - t_i = \Delta t$. Using finite-difference approximations for time and spatial derivatives:

$$\frac{\partial T(x_j,t)}{\partial t} = \frac{T_j(t_{i+1}) - T_j(t_i)}{\Delta t} \quad \text{and} \quad \frac{\partial^2 T(x,t)}{\partial x^2} = \frac{T_{j-1}(t_i) - 2T_j(t_i) + T_{j+1}(t_i)}{(\Delta x)^2},$$

for a replacement of derivatives in our continuous PDE, which provides one linear equation for each point x_j. Using explicit Euler method for time integration, we obtain, after some rearrangement of factors, a simple algorithm for calculation of new temperatures $T(t_{i+1})$ from old temperatures:

$$T_j(t_{i+1}) = T_j(t_i) + \frac{c\,\Delta t}{(\Delta x)^2}\left(T_{j-1}(t_i) - 2T_j(t_i) + T_{j+1}(t_i)\right).$$

In each discretization point, a new temperature $T_j(t_{i+1})$ is obtained by summing the old temperature with a product of a constant factor and linear combination of temperatures in three neighboring points. After we determine the initial temperatures of inner points and fix the temperatures in boundary points, we can start marching in time to obtain updated values of the bar temperature.

Note, that in the explicit Euler method, thermal conductivity c, spatial discterization Δx and time-step Δt must be in an appropriate relation that fulfil CFL stability condition, which is for our case: $c\,\Delta t/(\Delta x)^2 \leq 0.5$. The CFL condition could be informally explained with a fact that a numerical method has to step in time slower than the simulated physical phenomenon. In our case, the impact of a change in discretization point temperature is at most in neighboring discretization points. Hence, with smaller Δx (denser discretization points), shorter time-steps are required in order to correctly capture the simulated diffusion of temperature.

When we have to stop the iteration? Either after a fixed number of time-steps nt, or when the solution achieves a specified accuracy, or if the maximum difference between previous and current temperatures falls below a specified value. A sequential algorithm for an explicit finite differences solution of our PDE is provided below:

Algorithm 3 SEQUENTIAL ALGORITHM: 1-D_HEAT_EQUATION

Input: err - desired accuracy;
N - number of discretization points
T_0 - initial temperature
T_L and T_R - boundary temperatures

1: Discretize domain
2: Set initial solution vectors $\mathbf{T}_i$ and $\mathbf{T}_{i+1}$
3: **while** Stopping criteria NOT fulfilled **do**
4: **for** $j = 2 \ldots (N-1)$ **do**
5: Calculate new temperature $T_j(t_{i+1})$
6: **end for**
7: **end while**

Output: **T** - Approximate temperature solution in discretization points.

We start again with the validation of our program, on a notebook with a single MPI process and with the code from Listing 7.2. The test parameters were set as follows: $p = 1$, $N = 30$, $nt = [1, \ldots, 1000]$, $c = 9\mathsf{e} - 3$, $T_0 = 20$, $T_L = 25$, $T_R = 18$, $time = 60$, $L = 1$. Because the exact solution is known, i.e. a line between T_L, T_R, the maximal absolute error of the numerical solution was calculated to validate the solution behaviour. If the model and numerical method are correct, the error should converge to zero.

The resulting temperatures evolution in time, on the simulated bar, are shown in Fig. 7.1.

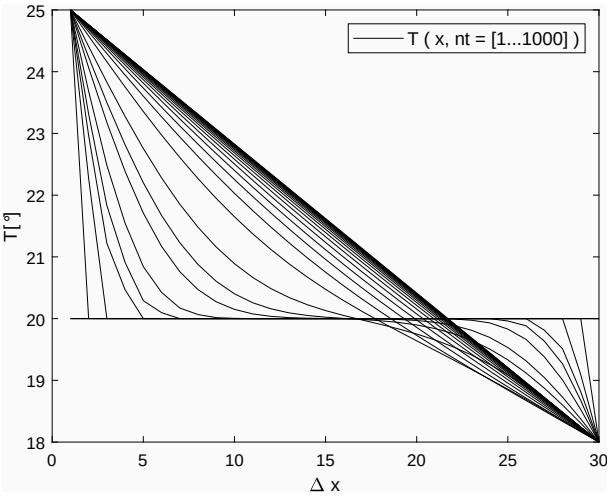


Fig. 7.1: Temperature evolution in space and time as solved by heat equation.

The set of curves in Fig. 7.1 confirms that the numerical method produces in initial time-steps a solution near to initial temperature. Then the simulated temper-

atures changes as expected. While the number of time-steps *nt* increases, the temperatures advance towards the correct result, which we know that is a straight line between left and right boundary temperatures, i.e. in our case, between 25° and 18°.

We have learned, that in the described solution methodology, the calculation of a temperature T_j in a discretization point depends only on temperatures in two neighbouring points, consequently, the algorithm has a potential to be parallelized. We will again use domain decomposition, however, the communication between processes will be needed in each time-step, because the left and right discretization point of a sub-domain are not immediately available for neighboring processes. A point-to point communication is needed to exchange boundaries of sub-domains. In 1-D case, the discretized bar is decomposed in a number of sub-domains, which is equal to the number of processes. All sub-domains should manage a similar number of points, because an even load balance is desired. Some special treatment is needed in calculation near domain boundaries.

Regarding the communication load, some collective communication is needed at the beginning and end of the calculation. Additionally, a point-to-point communication is needed in each time-step to exchange the temperature of sub-domain border points. In our simplified case, the calculation will stop after a predefined number of time-steps, hence, no communication is needed for this purpose. The parallelized algorithm is shown below:

Algorithm 4 PARALLEL ALGORITHM: 1-D_HEAT_EQUATION

Input: err - desired accuracy;
N - number of discretization points
T_0 - initial temperature
T_L and T_R - boundary temperatures

1: Get *myID* and the number of cooperating processes p
2: Master broadcast *Input* parameters to all processes
3: Set local solution vectors $\mathbf{T}_{ip}$ and $\mathbf{T}_{ip+1}$
4: Compute a shorter **for** loop:
5: **while** Stopping criteria NOT fulfilled **do**
6: **for** $j = 1 \ldots N/p$ **do**
7: Exchange $T_j(t_i)$ of sub-domain border points
8: Calculate new temperature $T_j(t_{ip+1})$
9: **end for**
10: Master gather sub-domain temperatures as a final result **T**
11: **end while**

Output: **T** - Approximate temperature solution in discretization points.

We see that in the parellelized program several tasks has to be implemented, i.e. user interface for input/output data, decomposition of domain, allocation of memory for solution variables, some global communication, calculation and local communication in each time-step, and assembling of the final result. Some of the tasks are inherently sequential and will therefore limit the speed-up, which will be more pronounced in smaller systems.

The processes are not identical. We will again use a master process and several slave processes, two of them responsible for domain boundary. The master process will manage input/output interface, broadcast of solution parameters and gathering of final results. The analysis of parallel program performances for OpenMP, MPI, are described in the following sections.

7.1 OpenMP

Computing 1-D heat transfer using a multicore processor is simple if Algorithm 3 is taken as the starting point for parallelization. As already explained above, iterations of the inner loop are independent (while the iterations of the outer loop must be executed one after another). The segment of the OpenMP program implementing Algorithm 3 is shown in Listing 7.1: `Told` and `Tnew` are two arrays where the temperatures computed in the previous and in the current outer loop iteration are computed; `C` contains the constant $c\Delta t/(\Delta x)^2$.

```
#pragma omp parallel firstprivate(k)
  {
    double *To = Told;
    double *Tn = Tnew;
    while (k--) {
      #pragma omp for
      for (int i = 1; i <= n; i++) {
        Tn[i] = To[i]
          + C * (To[i - 1] - 2.0 * To[i] + To[i + 1]);
      }
      double *T = To; To = Tn; Tn = T;
    }
  }
```

Listing 7.1: Computing heat transfer in one dimension.

It is woth examining the wall clock time of this algorithm first. Figure 7.2 summarizes the wall clock time needed for 10^6 iterations in 10^5 points using a quadcore processor with multithreading. With more than 4 threads nothing is gained in terms of a wall clock time. As this is floating point intensive application run on a processor with one floating point unit per physical core, this can be expected: two threads running running on two logical cores of the same pysical core must share the same floating point unit. This leads to more synchronizing among threads and consequently to the increase of the total CPU time as shown in Figure 7.3.

The interested reader might investigate how the wall clock time and the speedup change if the ratio between the number of points along the bar and the number of iterations in time change. Namely, decreasing the number of points along the bar makes the synchronization at the barrier at the end of the parallel `for` loop relatively more expensive.

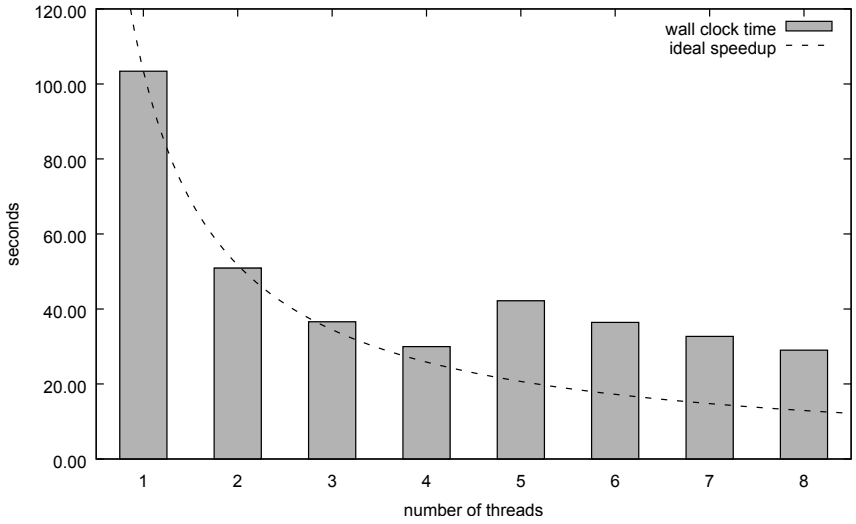


Fig. 7.2: The wall clock time needed to compute 10^6 iterations of 1-D heat transfer along a bar in 10^5 points.

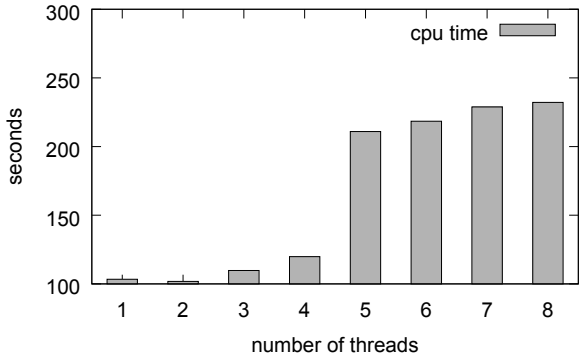


Fig. 7.3: The total CPU time needed to compute 10^6 iterations of 1-D heat transfer along a bar in 10^5 points.

7.2 MPI

In the solution of heat equation, the problem domain is discretized first. In our simplified case, a temperature diffusion in a thin bar is computed, which is modelled by 1-D domain. For efficient use of parallel computers the problem domain must be partitioned (decomposed) in possibly equal sub-domains. The number of sub-

domains should be equal to the number of MPI processes p. We prescribe a certain number of discretization points per process Np, which automatically guarantees a balanced computational load. Note, that the total number of discretization points N scales with the number of processes. In all active processes, an appropriate amount of memory is allocated for current and new solution vectors and initialized with initial and boundary values. Then, the CFL stability condition is verified.

In each time-step, every process that computes its sub-domain, exchanges sub-domain border temperatures with processes that compute its left end right sub-domains. In our implementation, blocking communication is used that can adequately maintain short messages. Then, new temperatures are calculated for all discretization points in each sub-domain, by methodology explained in the beginning of this chapter. We determine a fixed number of time-steps, therefore a special stopping criteria is not needed.

An exemplar MPI program implementation of a solution of 1-D heat equation for our simplified case is given in Listing 7.2.

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include "mpi.h"

void solve(int my_id, int num_p);

int main(int argc, char *argv[])
{
  int my_id, num_p;
  double start, end;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &my_id);
  MPI_Comm_size(MPI_COMM_WORLD, &num_p);
  if (my_id == 0)
    start = MPI_Wtime();
  solve(my_id, num_p);
  if (my_id == 0)
  printf("Elapsed seconds = %f\n", MPI_Wtime() - start);
  MPI_Finalize();
  return 0;
}
void solve(int my_id, int num_p)  //compute time step T
{
  double cfl, *T, *T_new;
  int i, j, tag, j_min = 0;
  int j_max = 10000;  //number of time-steps
  int N = 5000;  //number of points per process
  double c = 1e-11;  //diffusivity
  double T_0 = 20.0, T_L = 25.0, T_R = 18.0; //temperatures
  double time, time_new, delta_time, time_min = 0.0, time_max = 60.0;
  double *X, delta_X, X_min = 0.0, Length = 0.1;
  MPI_Status status;

  if (my_id == 0)
  {
    printf("  dT/dt = c * d^2T/dx^2  for %f < x < %f\n",X_min,Length);
    printf("  and %f < t <= %f.\n", time_min, time_max);
    printf("  space discretized by %d equidistant points \n",num_p*N);
    printf("  each processor works on %d points!!\n", N);
    printf("  time discretizted with %d equal time-steps.\n", j_max);
    printf("  number of cooperating processes is %d\n", num_p);
```

```
  }
  //allocate local buffers and calculate new temperatures
  X = (double *)malloc((N+2)*sizeof(double)); //N+2 point coordinates
  for (i = 0; i <= N + 1; i++)   //ghost points are in X[0] and X[N+1]
  {
    X[i] = ((double)(my_id * N + i - 1) * Length
      + (double)(num_p* N - my_id * N - i) * X_min)
      / (double)(num_p* N - 1);
  }
  T = (double *)malloc((N + 2) * sizeof(double));  //allocate
  T_new = (double *)malloc((N + 2) * sizeof(double));
  for (i = 1; i <= N; i++)
    T[i] = T_0;
  T[0] = 0.0; T[N + 1] = 0.0;
  delta_time = (time_max - time_min) / (double)(j_max - j_min);
  delta_X = (Length - X_min) / (double)(num_p* N - 1);

  cfl = c * delta_time / pow(delta_X,2); //check CFL
  if (my_id == 0)
    printf("  CFL stability condition value = %f\n", cfl);
  if (cfl >= 0.5)
  {
    if (my_id == 0)
      printf("  Computation cancelled: CFL condition failed.\n");
    return;
  }
  for (j = 1; j <= j_max; j++)  //compute T_new
  {
    time_new = ((double)(j - j_min) * time_max
      + (double)(j_max - j) * time_min)
      / (double)(j_max - j_min);
    if (0 < my_id)   //send T[1] to my_id-1 //replace with SendRecv?
    {
      tag = 1;
      MPI_Send(&T[1], 1, MPI_DOUBLE, my_id-1, tag, MPI_COMM_WORLD);
    }
    if (my_id < num_p - 1)   //receive T[N+1] from my_id+1.
    {
      tag = 1;
      MPI_Recv(&T[N+1],1,MPI_DOUBLE,my_id+1,tag,MPI_COMM_WORLD,&status)↩
    ;
    }
    if (my_id < num_p - 1)  //send T[N] to my_id+1
    {
      tag = 2;
      MPI_Send(&T[N], 1, MPI_DOUBLE, my_id+1, tag, MPI_COMM_WORLD);
    }
    if (0 < my_id)   //receive T[0] from my_id-1
    {
      tag = 2;
      MPI_Recv(&T[0],1,MPI_DOUBLE,my_id-1,tag,MPI_COMM_WORLD,&status);
    }
    for (i = 1; i <= N; i++)  //update temperatures
    {
      T_new[i] = T[i] + (delta_time * c/pow(delta_X,2)) *
                        (T[i-1] - 2.0 * T[i] + T[i+1]);
    }
    if (my_id == 0)  T_new[1] = T_L;  //update boundaries T with BC
    if (my_id == num_p- 1) T_new[N] = T_R;
    for (i = 1; i <= N; i++)  T[i] = T_new[i]; //update inner T
  }
  free(T);
  free(T_new);
  free(X);
  return;
}
```

Listing 7.2: Implementation of a solution of 1-D heat equation.

After the successful validation, already presented, the analysis of run-time behaviour, again on a two-core notebook computer and on eight cluster computers, has been performed. To fulfil the CFL condition, variables: t, nt, Np and c has to be appropriately selected. We set the number of all discretization points to $N = [5e5, 5e4, 5e3]$, therefore the number of points per process Np is obtained by scaling with the number of processes p. For example if $N = 5e4$ and $p = 4$, $Np = 1.25e5$, etc. For accurate timings and balanced communication and computation load nt was set to 1e4. To be sure about CFL condition, constant c is set to $1e-11$. Other parameters remain the same as in the PDE validation test. The parallel program run-time (RT) in seconds and speed-up (SU), as a function of the number of processes $p = [1, \ldots, 8]$ and discretization points, on a notebook computer, are shown in Fig. 7.4.

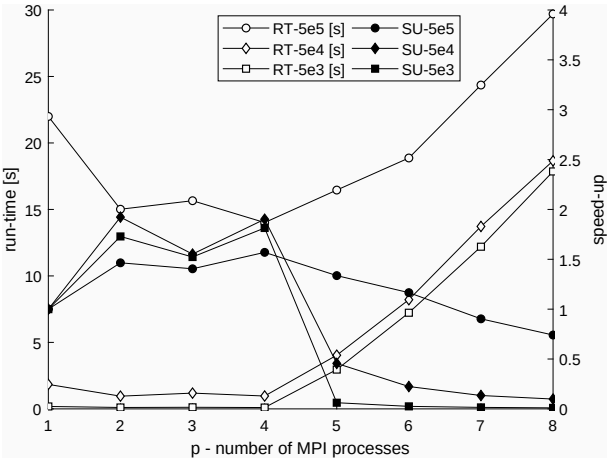


Fig. 7.4: Parallel run-time (RT) and speed-up (SU) of heat equation solution for $N = [5e5, 5e4, 5e3]$ and 1 to 8 MPI processes on a notebook computer.

The obtained results bring two important messages. First, the maximum speed-up is only about two, which is smaller than in the case of π calculation. The explanation for this could be in a smaller computation/communication time ratio. It seems that the program spent on communication almost the same time as on the calculation. Second, the speed-up drops significantly on more than 4 MPI processes, below one, which is even more pronounced with smaller number of discretization points. The explanation of such a behaviour is in the amount of communication, which is performed after each time-step.

Next experiments were performed on eight computing cluster nodes with the same approach as in Chapter 6 and with the same parameters as in the notebook

test. In this case we use `mpiexec` parameter `-bind-to core:1`, which appears to be more promising in previous tests. We can expect a high impact of communication load. Even that the messages are short, with just a few doubles, the delay is significant mainly because of the communication start-up time. The run-time (RT-B) in seconds and corresponding speed-up (SU-B), as a function of the number of MPI processes $p = [1, \ldots, 128]$ and the number of discretization points $N = [5e5, 5e3]$ is shown in Fig. 7.5.

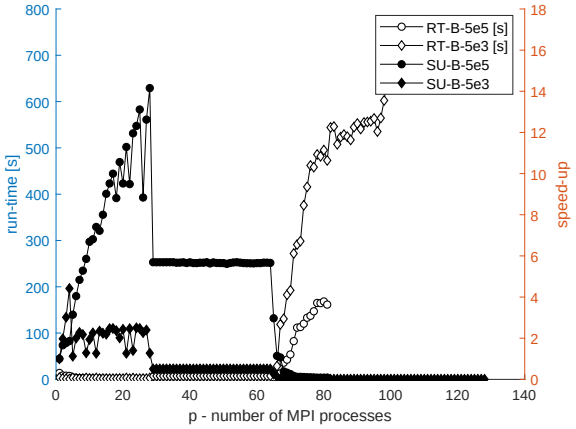


Fig. 7.5: Parallel run-time (RT-B) and speed-up (SU-B) of heat equation solution for $N = [5e5, 5e3]$ and 1 to 128 MPI processes bound to cores.

The results are a surprise! The speed-up is very unstable and approaches to maximum value 14 with about 30 processes. Then it jumps to 6 and remain stable until 64 processes with another jump to almost 0, with more than 64 processes. We guess that the problem is in communication.

First step occurs when the number of MPI processes increases from 36 to 37. This step happens because one communication channel gets additional burden. When the number of processes is 36 or lower, only a single pair of processes per neighbouring node is communicating among themselves. When process number 37 is created, it is assigned to cluster node 2 and requires communication with the process number 36 on node 1. This communication between nodes 1 and 2 is in addition to the communication required by processes number 4 and 5, which are also assigned to nodes 1 and 2, which could explain that the communication between nodes 1 and 2 take twice longer than before. Such a behaviour slow-down the whole program becose the remaining processes are waiting.

Then processes from number 38 to 64 are only adding to communication burden of other neighboring nodes, which happens in parallel and therefore does not additionally degrade the performance. Since the communication overhead at this number

of MPI processes easily overwhelms calculation, the speedup seems constant from there on.

Second step happens, when number of processes passes 64. Process 65 is again assigned to node 1. This makes it the ninth MPI process allocated on this node, which only supports 8 threads. 9-th and 1-st process on node 1 therefore have to share to the same core, which seems to be the recipe for abysmal performance. The speed-up drop is so overwhelming at this point because MPI uses busy waiting as a part of its synchronous send and receive operations. Busy waiting means that processes are not immediately switched when waiting for MPI communication, since the operating systems does not see them as idle but rather as busy. Therefore the waiting times on MPI communication dramatically increase and with them the execution times.

Therefore we repeat the experiment again. Now the processors are connected in a true physical ring topology with two communication ports per processor. Additionally, we replace the `MPI_Send` and `MPI_Recv` pairs by `MPI_Sendrecv` function. We reduce the number of processes in this experiment to 64, because larger numbers have been proved as useless. The run-time (RT-R) in seconds and corresponding speed-up (SU-R), as a function of the number of MPI processes $p = [1,\ldots,65]$ and the number of discretization points $N = [5\text{e}5, 5\text{e}3]$ is shown in Fig. 7.6.

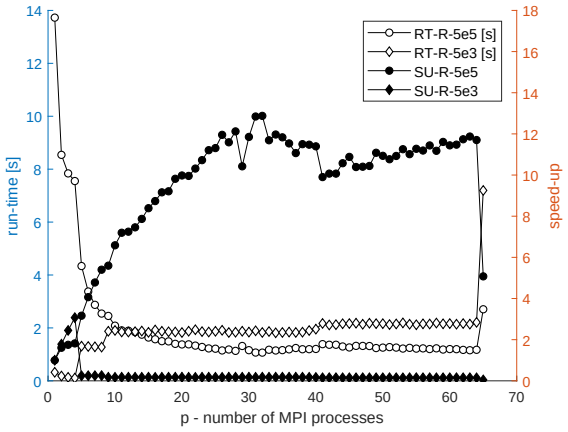


Fig. 7.6: Parallel run-time (RT-R) and speed-up (SU-R) of heat equation solution for $N = [5\text{e}5, 5\text{e}3]$ and 1 to 65 MPI processes on a ring interconnection topology and implemented by `MPI_Sendrecv`.

We can notice several improvements now. With larger number of discretization points, the speed-up is quite stable but the maximum is not higher than in previous experiment from Fig. 7.5. With smaller number of discretization point a speed-up is detected only in up to four processes, because a local memory communication is used, which confirms that the communication load is prevailing in this case. Further

investigation needs a lot of exciting engineering work, however, it is beyond the scope of this book and is left to enthusiastic readers.

Chapter 8
Engineering: Parallel implementation of Seam Carving

Seam-carving is a content-aware image resizing technique where the image is reduced in size by one pixel of width (or height) at a time. Seam carving attempts to reduce the size of a picture while preserving the most interesting content of the image. Seam Carving was originally published in a 2007 paper by Shai Avidan and Ariel Shamir. Ideally, one would remove the 'lowest energy' pixels (where energy means the amount of important information contained in a pixel) from the image to preserve the most important features. However, that would create artefacts and not preserve the rectangular shape of the image. To balance removing low energy pixels while minimizing artefacts, we remove exactly one pixel in each row (or column) where every pixel in the row must touch the pixel in the next row either via an edge or corner. Such a connected path of pixels is called seam. If we are going to resize the image horizontally, we need to remove one pixel from each row of the image. Our goal is to find a path of connected pixels from the bottom of the image to the top. By 'connected', we mean that we will never jump more than one pixel left or right as we move up the image from row to row. A **vertical seam** in an image is a path of pixels connected from the top to the bottom with one pixel in each row. Each row has exactly only one pixel which is the part of the vertical seam. By removing the vertical seams iteratively, we can compress the image in the horizontal direction. The seam carving method produces a resized image by searching for the seam which has the lowest user-specified 'image energy'. To shrink the image, the lowest energy seam is removed and the process is repeated until the desired dimensions are reached. Seam Carving is a three-step process:

1. Assign an energy value to every pixel. This will define the important parts of the image that we want to preserve.
2. Find an 8-connected path of the pixels with the least energy. We use dynamic programming to calculate the costs of every potential path through the image.
3. Follow the cheapest path to remove one pixel from each row or column to resize the image.

Following these steps will shrink the image by one pixel. We can repeat the process as many times as we want to resize the image as much as necessary. What we need

to is to implement a function which takes an image as an input and produce a resized image in one dimension or two dimensions as an output which is expected by the users.

Why is seam carving interesting for us? Effective parallelization of seam carving is a challenging problem due to its complex computation model. There are two main reasons why effective parallelization is prevented: (1) Computation dependence: dynamic programming is a key step to compute an optimal seam during image resizing and takes a large fraction of the program execution time. It is very hard to parallelize the dynamic programming on GPU devices due to the computation dependency. (2) Intensive and irregular memory access: in order to compute various intermediate results a large number of irregular memory access patterns is required. This worsens the program performance significantly. In this chapter, we are not going to find or present a better algorithm for seam carving that can be parallelized. We are just going to show, which part of the original seam carving algorithm can be accelerated on GPU and which parts cannot and how this affects the overall performance. For illustration purposes, we will use the *cyclist* image from Figure 8.1 as an input image, which is to be made narrower with seam carving.

Fig. 8.1: Original *cyclist* image to be made narrower with seam carving.

8.1 Energy calculation

What are the most important parts of an image? Which parts of a given image should we eliminate first when resizing and which should we hold onto the longest? The answer to these questions lies in the energy value of each pixel. The energy value of a pixel is the amount of important information contained in that pixel. So, the

first step is to compute the energy value for every pixel, which is a measure of its importance - the higher the energy, the less likely that the pixel will be included as part of a seam and eventually removed.

The simplest and frequently most-effective measure of energy is the gradient of the image. An image gradient highlights the parts of the original image that are changing the most. It is calculated by looking at how similar each pixel is to its neighbours. Large uniform areas of the image will have a low gradient (i.e. energy) value and the more interesting areas (edges of objects or places with a lot of detail) will have a high energy value. There exist a variety of energy measures (e.g. Sobel operator). In this book, which is not primarily devoted to image processing, we will use a very simple energy function, although a number of other energy functions exist and may work better. Let each pixel (i, j) in the image has its color denoted as $I(i, j)$. The energy of the pixel (i, j) is given by the following equation:

$$E(i,j) = |I(i,j) - I(i,j+1)| + |I(i,j) - I(i+1,j)| + |I(i,j) - I(i+1,j+1)| \quad (8.1)$$

A sequential algorithm for pixel energy calculation is given in Algorithm 5.

Algorithm 5 SEAM CARVING: ENERGY CALCULATION

Input: I - *RxC* image

1: **for** $i = 1 \ldots R$ **do**
2: **for** $j = 1 \ldots C$ **do**
3: $E(i,j) = |I(i,j) - I(i,j+1)| + |I(i,j) - I(i+1,j)| + |I(i,j) - I(i+1,j+1)|$
4: **end for**
5: **end for**

Output: E - *RxC* energy map

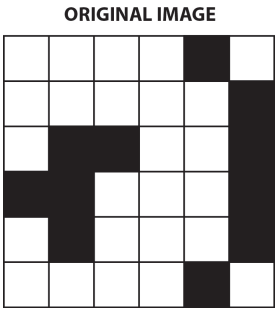
We will illustrate this step on a simple example. Let's suppose a black and white image as in Figure 8.2a and let's suppose that the color of black pixels is coded with the value 1, and the color of white pixels is coded with 0. Figure 8.2b shows energy map for the image from Figure 8.2a. Energies of the pixels in the last column and the last row are computed with assumption that the image is zero-padded. For example, the energy of the first pixel in the fourth row (black pixel) is according to Equation 8.1:

$$E(3,0) = |1-1| + |1-0| + |1-1| = 1,$$

and the energy of the last pixel in the fifth row is:

$$E(4,5) = |1-0| + |1-0| + |1-0| = 3.$$

The resulting *cyclist* image of this step is shown in Figure 8.3. We can see that large uniform areas of the image have a low gradient value (black) and the more interesting edges of objects have a high gradient value (white).



(a)

STEP 1

0	0	0	1	2	1
1	1	0	0	2	2
3	1	2	0	2	2
1	2	0	0	2	2
1	3	0	1	2	3
0	0	0	1	2	0

(b)

Fig. 8.2: (a) Original black and white image. (b) Energies of the pixels in the image.

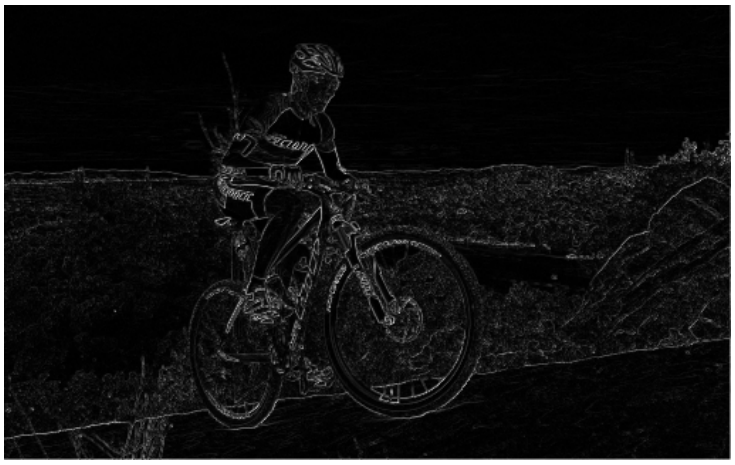


Fig. 8.3: The calculated energy function of the *cyclist* image.

8.2 Seam identification

Now that we have calculated the value of each pixel, our next objective is to find a path from the bottom of the image to the top of the image with the least energy. The line must be 8-connected: this means that every pixel in the row must be touched by the pixel in the next row either via an edge or corner. That would be a vertical seam of minimum total energy. One way to do this would be to simply calculate the costs of each possible path through the image one-by-one. Start with a single pixel in the bottom row, navigate every path from there to the top of the image, keep track of the cost of each path as you go. But we will end up with thousands or millions of possible paths. To overcome this, we can apply the dynamic programming method as described in the paper by Avidan and Shamir. Dynamic programming lets us touch each pixel in the image only once, aggregating the total cost as we go, in order to

calculate the final cost of an individual path. Once we have found the energy of every pixel, we start at the bottom of the image and go up row by row, setting each element in the row to the energy of the corresponding pixel plus the minimum energy of the 3 possibly path pixels 'below' (the pixel directly below and the lower left and right diagonal). Thus, we have to traverse the image from the bottom row to the first row and compute the cumulative minimum energy M for all possible connected seams for each pixel (i, j):

$$M(i,j) = E(i,j) + min(M(i+1,j-1), M(i+1,j), M(i+1,j+1)) \tag{8.2}$$

In the bottom most row cumulative energy is equal to pixel energy, i.e. $M(i,j) = E(i,j)$. A sequential algorithm for cumulative energy calculation is given in Algorithm 6.

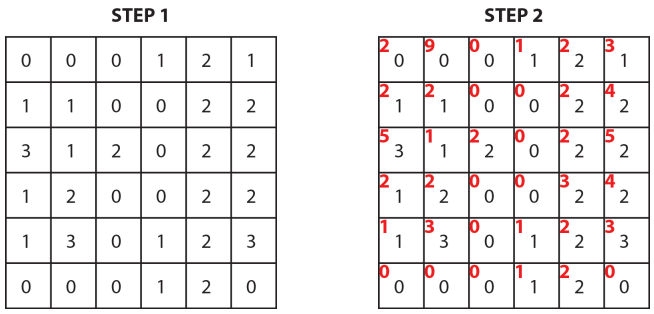
Algorithm 6 SEAM CARVING: CUMULATIVE ENERGY CALCULATION

Input: E - *RxC* energy map

1: **for** $j = 1 \ldots C$ **do**
2: $M(R,j) = E(R,j)$
3: **end for**
4: **for** $i = R-1 \ldots 1$ **do**
5: **for** $j = 1 \ldots C$ **do**
6: $M(i,j) = E(i,j) + min(M(i+1,j-1), M(i+1,j), M(i+1,j+1))$
7: **end for**
8: **end for**

Output: M - *RxC* cumulative energy map

We will illustrate this step with Figure 8.4. We start with the last (bottom-most) row. Cumulative energies in that row are the same as pixel energies. Then we move up to the fifth row. Cumulative energy of the fifth pixel in the fifth row is the sum of



(a) (b)

Fig. 8.4: (a) Energies of the pixels. (b) Cumulative energies.

its energy and the minimal energy of three pixels below it:

$$M(4,4) = 2 + min(1,2,0) = 2.$$

On the other hand, cumulative energy of the last pixel in the fifth row is:

$$M(4,5) = 3 + min(2,0,\infty) = 3.$$

As the element $(4,6)$ does not exist, we assume it has the maximal energy. In other words, we ignore it. Once we have computed all the values M, we simply find the lowest value of M in the top row and return the corresponding path as our minimum energy vertical seam.

This effect of this step is easy to see in Figure 8.5. Notice how the spots where the

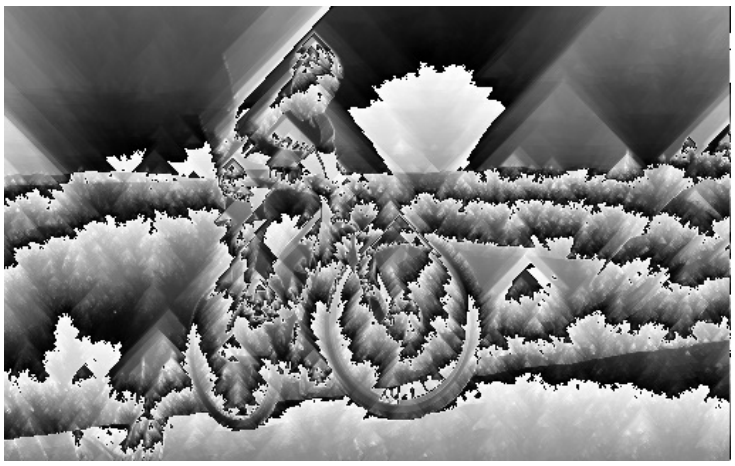


Fig. 8.5: The calculated cumulative energy function of the *cyclist* image. Please note that due to summation, almost all pixel values are greater than 255 and are represented only with least significant eight bits in the image.

gradient image was brightest are now the roots of inverted triangles of brightness as the cost of those pixels propagate into all of the pixels within the range of the widest possible paths upwards. For example, the brightest inverted triangle at the center of the image (in from of the cyclist) is created because the white edge at horizon propagates upwards. When we arrive at the top row, the lowest-valued pixel will necessarily be the root of the cheapest path through the image. Now we are ready to start removing seams.

8.3 Seam labeling and removal

The final step is to remove all of the pixels along the vertical seam. Due to the power of dynamic programming, the process of actually removing seams is quite easy. All we have to do to calculate the cheapest seam is to start with the lowest value M in the top row and work our way up from there, selecting the cheapest of the three adjacent pixels in the row below. Dynamic programming guarantees that the pixel with the lowest value M will be the root of the cheapest connected path from there. Once we have selected which pixels we want to remove, all that we have to do is go through and copy the remaining pixels on the right side of the seam from right to left and the image will be one pixel narrower. A sequential algorithm for seam removal is given in Algorithm 7.

Algorithm 7 SEAM CARVING: SEAM REMOVAL

Input: M - *RxC* cumulative energy map
Input: I - *RxC* original image

```
 1: min = M(1,1)
 2: col = 1
 3: for j = 2...C do
 4:     if M(1,j) < min then
 5:         min = M(1,j)
 6:         col = j
 7:     end if
 8: end for
 9: for i = 1...R do
10:     for j = col...C do
11:         I(i,j) = I(i,j+1)
12:     end for
13:     if M(i+1,col-1) < M(i+1,col) then
14:         col = col-1
15:     end if
16:     if M(i+1,col+1) < M(i+1,col-1) then
17:         col = col+1
18:     end if
19: end for
```

Output: I - *RxC* resized image

We will illustrate this step with Figure 8.6. We start with the top most row and find the pixel with the smallest value M. In our case this is the third pixel in the first row with $M(0,2) = 0$. Then we select the pixel below that one with the minimal M. In our case this is the third pixel in the second row with $M(1,2) = 0$. We continue downwards and select the pixel bellow the current with the minimal cumulative energy. This is the fourth pixel in the third row with $M(2,3) = 0$. We continue this process until the last row. The seam with minimal energy is depicted in grey in Figure 8.6b.

Figure 8.7 shows the labeled (with white pixels) seams in the *cyclist* image. The

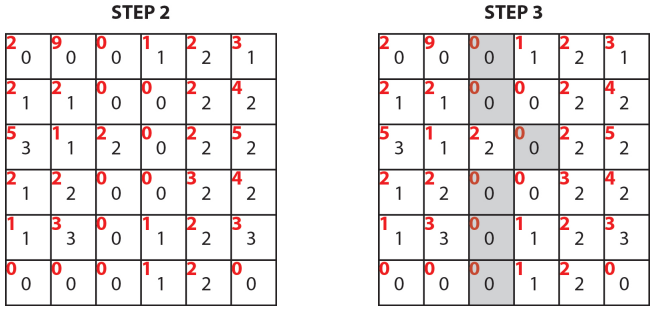


Fig. 8.6: (a) Cumulative energies. (b) The seam (in grey) with the minimal energy.

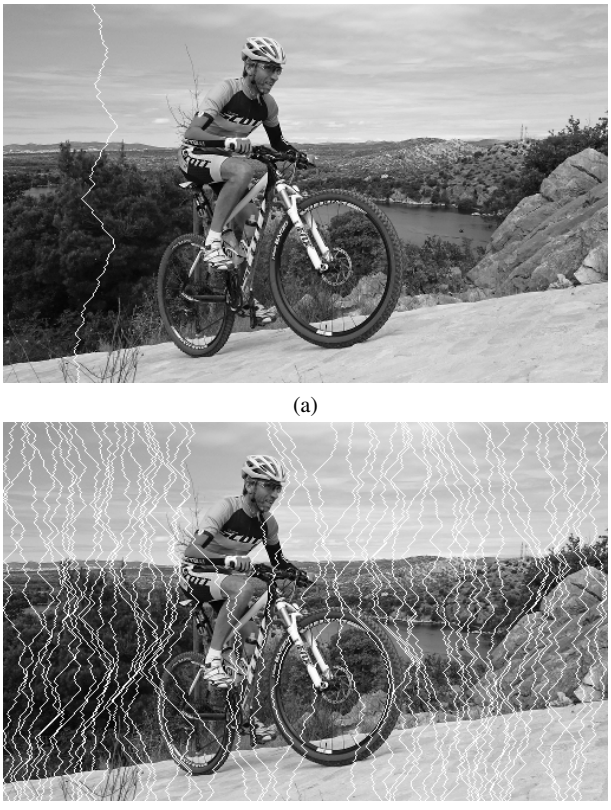


Fig. 8.7: (a) The first seam in the *cyclist* image. (b) The first 50 seams in the *cyclist* image.

very first vertical seam found in the *cyclist* image is depicted in Figure 8.7a. It goes through the darkest parts of the energy map form Figure 8.3 and thus through the pixels with minimal amount of information. Figure 8.7b depicts the first 50 seams found in the *cyclist* image. It can be observed how seams are 'avoiding' the regions with the highest pixel energy and thus the highest amount of information.

Once we have labeled a vertical seam we go through the image and move the pixels that are located at the right of the vertical seam from right to left. The new image would be one pixel narrower than the original. We repeat the whole process for as many seams as we like to remove. Figure 8.8 shows two *cyclist* images reduced in size by (a) 100 pixels and (b) 350 pixels.

(a) (b)

Fig. 8.8: (a) The image resized by removing 100 seams. (b) The image resized by removing 350 seams.

8.4 Seam carving on GPU

In this section we present and compare the implementations of seam carving on CPU and GPU.

8.4.1 Seam carving on CPU

We will first present the CPU code for seam carving. Emphasis will be only on the functions that implement the main operations of the seam carving algorithm. Other helper functions and code are available on the book's companion site.

Energy calculation on CPU

Listing 8.1 shows how to calculate pixel energy following Algorithm 5. The function `simpleEnergyCPU` reads input PGM image `input`, calculates the energy for every pixel and writes the energy to the corresponding pixel in PGM image `output`.

```
void simpleEnergyCPU(PGMData *input, PGMData *output, int new_width)
{
    int i, j;
    int diffx, diffy, diffxy;
    int tempPixel;

    for(i=0; i<(input->height); i++)
        for(j=0; j<new_width; j++)
        {
            diffx = abs(getPixelCPU(input, i, j) -
                        getPixelCPU(input, i, j+1));
            diffy = abs(getPixelCPU(input, i, j) -
                        getPixelCPU(input, i+1, j));
            diffxy = abs(getPixelCPU(input, i, j) -
                         getPixelCPU(input, i+1, j+1));

            tempPixel = diffx + diffy + diffxy;
            if( tempPixel>255 )
                output->image[i*output->width+j] = 255;
            else
                output->image[i*(output->width)+j] = tempPixel;
        }
}
```

Listing 8.1: Compute pixel energy.

The function Listing 8.1 implements image gradient, which highlights the parts of the original image that are changing the most. The image gradient is calculated using Equation 8.1, which looks at how similar each pixel is to its neighbors. The third argument `new_width` keeps track of the current image width.

Cumulative energies on CPU

Now that we have calculated the energy value of each pixel, our goal is to find a path of connected pixels from the bottom of the image to the top. As we previously said, we are looking for a very specific path: the one who's pixels have the lowest total value. In other words, we want to find the path of connected pixels from the bottom to the top of the image that touches the darkest pixels in our gradient image. Listing 8.2 shows how to calculate cumulative pixel energy following Algorithm 6. The function `cumulativeEnergiesCPU` reads input PGM image `input`, which contains the energy of pixels, and writes the cumulative energy to the corresponding pixel in PGM image `output`.

```
void cumulativeEnergiesCPU(PGMData *input, PGMData *output, int ↩
    new_width){
    //Start from the bottom-most row:
    for(int i = input->height-2; i >= 0; i--){
```

```
        for(int j = 0; j < new_width; j++){
            output->image[i*(input->width) + j] =
                input->image[i*(input->width) + j] +
                    getPreviousMin(output, i, j, new_width);
        }
    }
}
```

Listing 8.2: Compute cumulative energies.

Using dynamic programming approach, we start at the bottom and work our way up, adding the cost of the cheapest below neighbor to each pixel. This way, we accumulate cost as we go - setting the value of each pixel not just to its own cost, but to the full cost of the cheapest path from there to the bottom of the image. The helper function `getPreviousMin()` returns the minimal energy value from the previous row. It contains a few compare statments to find the minimal value. As we can see, each iteration in the outermost loop depends on the results from the previous iterations, so it cannot be parallelized only the iterations in the innermost loop are mutually independent and can be run concurrently. Also, to find the minimal value from the previous row, we should use conditional statements in the helper function `getPreviousMin()`. We already know that these statements will prevent the work-items to follow the same execution path and thus it will prevent effective execution of warps.

Seam labeling and removal on CPU

The process of labeling and removing a seam with minimal energy is quite easy. All we have to do to is to start with the darkest pixel with minimal cumulative energy in the top row and work our way down from there, selecting the cheapest of the three adjacent pixels in the row below and changing the color of the corresponding pixel in the original image to white. Listing 8.3 shows how to color the seam with minimal energy and Listing 8.4 shows how to remove the seam with minimal energy.

```
void seamIdentificationCPU(PGMData *input, PGMData *output, int ↩
    new_width){
    int column = 0;
    int minvalue = input->image[0];
    //find the minimum in the topmost row (0) and return column index:
    for(int j = 1; j < new_width; j++){
        if (input->image[j] < minvalue) {
            column = j;
            minvalue = input->image[j];
        }
    }

    //Start from the top-most row :
    for(int i = 0; i < input->height; i++){
        output->image[(i)*(input->width) + column] = 255;
        column = getNextMinColumn(input, i, column, new_width);
    }
}
```

Listing 8.3: Labeling the seam with the minimal energy.

```
void seamRemoveCPU(PGMData *input, PGMData *output, int new_width){
    int column = 0;
    int minvalue = input->image[0];
    //find the minimum in the topmost row (0) and return column index:
    for(int j = 1; j < new_width; j++){
        if (input->image[j] < minvalue) {
            column = j;
            minvalue = input->image[j];
        }
    }

    //Start from the top-most row:
    for(int i = 0; i < input->height; i++){
        // make this row narrower:
        for(int k = column; k < new_width; k++){
            output->image[i*(input->width) + k] =
                output->image[i*(input->width) + k+1];
        }
        column = getNextMinColumn(input, i, column, new_width);
    }
}
```

Listing 8.4: Seam removal.

We can see in Listing 8.4 that seam removal starts with the loop in which we locate the pixel with minimal cumulative energy, i.e. the first pixel in the vertical seam with the minimal energy. Then we proceed to the loop nest. The outermost loop indexes rows in the image. In each row we remove the seam pixel - we move the pixels that are located at the right of the vertical seam from right to left. After that, we have to find the column index of the seam pixel in the next row. We do this using the helper function `getNextMinColumn()`. As in the previous step, we use conditional statements in the helper function `getPreviousMin()`, which will prevent the work-items to follow the same execution path.

8.4.2 Seam carving in OpenCL

In this subsection we will discuss the possible implementation of seam carving on GPU. The host code would be responsible for the following steps:

1. Load image from file into a buffer. For example, we can use greyscale PGM images, which are easy to handle.
2. Transfer the image in buffer to the device.
3. Execute four kernels: the energy calculation kernel, the cumulative energy kernel, the seam labeling kernel and the seam removal kernel.
4. Read the resized image from GPU.

As discussed before, seam carving consists of three steps. We will implement each step as one ore more kernel functions. The complete OpenCL code for seam carving can be found on the book's companion site.

OpenCL kernel function: Energy calculation

The first step of seam carving is embarrassingly (or perfectly) parallel because the calculation of the energy of individual pixels is completely independent of the energy of adjacent pixels. Each work-item will calculate the energy of the pixel whose index is the same as its global index. Energy is calculated from the values of adjacent pixels. Depending on the energy function used, we may need three, eight or even 24 adjacent pixels. Here we will use our simple energy function from Equation 8.1. The parallel algorithm for the energy calculation is given in Algorithm 8.

Algorithm 8 SEAM CARVING: PARALLEL ENERGY CALCULATION

Input: I - *RxC* image

1: **for all** work-item(i,j) in 2-dim NDRange **do**
2: $E(i,j) = |I(j,j) - I(rowGID, j+1)| + |I(i,j) - I(i+1,j)| + |I(i,j) - I(i+1,j+1)|$
3: **end for**

Output: E - *RxC* energy map

Listing 8.5 shows the code for the energy calculation kernel.

```
__kernel void simpleEnergyGPU(
                        __global int* imageIn,
                        __global int* imageOut,
                        int width,
                        int height,
                        int new_width) {

    // global thread index
    int columnGID = get_global_id(0);    // column in NDRange
    int rowGID = get_global_id(1);       // row in NDRange
    int tempPixel;
    int diffx, diffy, diffxy;

    diffx = abs_diff(imageIn[rowGID * width + columnGID],
                imageIn[rowGID * width + columnGID  + 1]);
    diffy = abs_diff(imageIn[rowGID * width + columnGID],
                imageIn[(rowGID+1) * width + columnGID]);
    diffxy = abs_diff(imageIn[rowGID * width + columnGID],
                imageIn[(rowGID+1) * width + columnGID + 1]);
    tempPixel = diffx + diffy + diffxy;

    if( tempPixel>255 )
        imageOut[rowGID*width+columnGID] = 255;
    else
        imageOut[rowGID*width+columnGID] = tempPixel;
}
```

Listing 8.5: The energy calculation kernel

Each pixel will also affect the energy of other adjacent pixels, so its will also be read by other work-items from the global memory. That means that the same word from the global memory will be accessed multiple times. Therefore, it makes sense to first load a block of pixels and their neighbours into the local memory and only

then start the calculation of energy. The reader should add the code for collaborative loading of the pixel block into local memory. Do not forget to wait for other work-items at the barrier before start to calculate the pixel energy.

OpenCL kernel function: Seam identification

Prior to calculating the cumulative energy of the pixels in one row, we should have already calculated the cumulative energy of all the pixels in the previous row. Because of this data dependency we can only run as many work-items at a time as the number of pixels in one row. When all work-items finish the calculation of the cumulative energy in one row, they move on to the next row. One work-item will calculate the cumulative energy of all pixels in the same column, but it will move to the next row (pixel above) only when all other work-items have finished the computation in the current row. Therefore, we need a way to synchronize work-items that calculate cumulative energies. We can synchronize work-items in two different ways:

1. We can run all work-items in the same (only one) work-group. The advantage of this method is that we can synchronize all work-items using barriers. The disadvantage of this approach lies in the fact that only one block of work-items can be run, so only one compute unit on GPU will be active during this step. In this approach, we will enqueue one kernel. The parallel algorithm for the cumulative energy calculation is given in Algorithm 9.

Algorithm 9 SEAM CARVING: PARALLEL CUMULATIVE ENERGY CALCULATION IN ONE WORK-GROUP

Input: E - *RxC* energy map

```
1: for all work-item(j) in 1-dim NDRange do
2:     M(R, j) = E(R, j)
3:     for i = R − 1 ... 1 do
4:         M(i, j) = E(i, j) + min(M(i+1, j−1), M(i+1, j), M(i+1, j+1))
5:         barrier()
6:     end for
7: end for
```

Output: M - *RxC* cumulative energy map

2. We can organize work-items in more than one work-group. The disadvantage of this approach is that work-groups cannot be synchronized with each other using barriers. But we can synchronize blocks by running one kernel at a time. One kernel, consisting of several work-groups will compute cumulative energies in just one row. When finished, we will have to rerun the same kernel, but with different arguments (i.e., the address of the new row). Thus we will enqueue the same kernel in a loop from the host code.

Which of two presented approaches is more appropriate depends on the size of the problem. For smaller images, the first approach may be more appropriate, while the second approach is more appropriate for images with very long rows since we can employ more compute units.

Listing 8.6 shows the code for the cumulative energy calculation kernel, which will be used for testing purposes in this book. The kernel does not use local memory and does not implement collaborative loading. The reader should implement this functionality and compare both kernels in terms of execution times. The reader should also implement the kernel for the second approach and measure the execution time.

```
__kernel void cumulativeEnergiesGPU(
                        __global int* imageIn,
                        __global int* imageOut,
                        int width,
                        int height,
                        int new_width) {

    // global thread index
    int columnGID = get_global_id(0);    // column in NDRange

    //Start from the bottom-most row:
    for(int i = height-2; i >= 0; i--){
        imageOut[i*width+columnGID] = imageIn[i*width+columnGID] +
                                getPreviousMinGPU(imageOut, i, ↩
    columnGID,
                                                        width, height, ↩
    new_width);
        // Synchronise to make sure the tiles are loaded
        barrier(CLK_LOCAL_MEM_FENCE);
    }
}
```

Listing 8.6: Cumulative energy calculation kernel - one work-group approach

OpenCL kernel function: Seam labeling and removal

The last step is to label and remove the seam with the minimal energy. Unfortunately, this step is inherently sequential because each pixel position in the seam strongly depends on the position of the previous pixel in the seam. So we cannot label all pixel in the seam in parallel. How can we implement this sequential function as a kernel on a massively parallel computer such as GPU? One possible solution would be that only one work-item labels and removes the whole seam. So the function for the GPU kernel would be almost the same as the function in Listing 8.3. The NDrange would have the dimension (1,1), i.e. we run only one work-item in the NDrange.

The better solution would be to divide this step into to operations: seam labeling and seam removal. The first step (seam labeling) should return column indices of every pixel in the seam. The kernel for this step will run in NDrange of dimension (1,1), i.e. only one work-item labels the whole seam. While the first step is inherently

sequential, the second could be parallelized as follows. We run as many work-items as number of rows in the input image. Each work item removes the seam pixel in its row and copies the remaining pixels on its right one position to the left. The second step uses the array of column indices from the first step to locate its seam pixel. A parallel algorithm for seam removal is given in Algorithm 10.

Algorithm 10 SEAM CARVING: SEAM REMOVAL

Input: I - *RxC* original image
Input: C - 1*xR* vector of column indices

1: **for all** work-item(i) in 1-dim NDrange **do**
2: column = C(i)
3: **for** $j = col \ldots C$ **do**
4: $I(i, j) = I(i, j+1)$
5: **end for**
6: **end for**

Output: I - *RxC* resized image

Listing 8.7 shows the code for the seam labeling kernel and Listing 8.8 shows the code for the seam removal kernel.

```
__kernel void getSeamGPU(
                __global int* imageIn,
                __global int* seamColumns,
                int width,
                int height,
                int new_width) {

    int column = 0;
    int minvalue = imageIn[0];

    //find the minimum in the topmost row (0) and
    //   return column index:
    for(int j = 1; j < new_width; j++){
        if (imageIn[j] < minvalue) {
            column = j;
            minvalue = imageIn[j];
        }
    }

    //Start from the top-most row:
    for(int i = 0; i < height; i++){
        column = getNextMinColumnGPU(imageIn, i,
                column, width,
                height, new_width);
        seamColumns[i] = column;
    }
}
```

Listing 8.7: Seam labeling kernel.

```
__kernel void seamRemoveGPU(
                        __global int* imageIn,
                        __global int* imageOut,
```

```
4                           __global int* seamColumns,
5                           int width,
6                           int height,
7                           int new_width) {
8
9       int iGID = get_global_id(0);       // row in NDRange
10
11      // get the column index of the seam pixel in my row:
12      int column = seamColumns[iGID];
13
14      // make my row narrower:
15      for(int k = column; k < new_width; k++){
16          imageOut[iGID*width + k] = imageOut[iGID*width + k+1];
17      }
18  }
```

Listing 8.8: Seam removal kernel.

To analyze the performance of the seam carving program, the sequential version has been run on a quadcore processor Intel Core i7 6700HQ running at 2,2 GHz, while the parallel version has been run on an Intel Iris Pro 5200 GPU running at 1,1 GHz. This is a small GPU integrated on the same chip as the CPU and has only 40 processing elements. The results of seam carving for an image of size 512x320 are presented in Table 8.1.

Table 8.1: Experimental results

Step	**CPU time [s]**	**GPU time [s]**	**Speedup**
Energy calculation	0.010098	0.000698	14.46
Cumulative energy	0.004696	0.003276	1.43
Seam removal	0.000601	0.005690	0.11
Total	**0.014314**	**0.009664**	**1.48**

As can be seen from the measured execution times, noticeable acceleration is achieved only for the first step. Although this step is embarrassingly parallel, we do not achieve the ideal speedup. The reason is that when calculating the energies of individual pixels, the work-items irregularly access the global memory and there is no memory coalescing. The execution times could be reduced if the work-items used local memory, as we did in matrix multiplication.

At the second step, the speedup is barely noticeable. The first reason for this is the data dependency between the individual rows. The other reason is, as before, irregular access to global memory. And the third factor that prevents effective parallelization is the usage of conditional statements when searching for minimal elements in previous rows. Here too, the times would be improved by using local memory.

At the third step, we do not even get speed up, but almost a 10X slowdown! The reason for such a slowdown lies in the fact that only one thread can be used to mark the seam.

And last but not least, the processing elements on the GPU runs at a 2X lower frequency than the CPU.

Chapter 9
Final remarks and perspectives

Now that we have come to the end of the book the reader should be well aware and informed that **parallelism is ubiquitous** in computing; it is present in hardware devices; in computational problems; algorithms; and in software on all levels.

Consequently, **many opportunities** for improving the efficiency of parallel programs are ever present. For example, theoreticians and scientists can search for and design new, improved parallel algorithms; programmers can develop better tools for compiling and debugging parallel programs; and cooperation with engineers can lead to faster and more efficient parallel programs and hardware devices.

Being so, it is our hope that our book will serve as the first step of a reader who wishes to join this ever evolving journey. We will delighted if the book will also encourage the reader to delve further in the study and practice of parallel computing.

As the reader now knows, the book provides many basic insights into parallel computing. It focuses on three main parallel platforms, the multi-core computers, the distributed computers, and the massively parallel processors. In addition, it explicates and demonstrates the use of the three main corresponding software libraries and tools, the OpenMP, the MPI, and the OpenCL library. Furthermore, the book offers hands-on practice and miniprojects so that the reader can gain experience.

After reading the book the reader may have become aware of the following three *general facts about the libraries* and their use on parallel computers:

- OpenMP is relatively easy to use yet limited with the number of cooperating computers.
- MPI is harder to program and debug but—due to the excellent support and long tradition—manageable and not limited with number of cooperating computers.
- Accelerators, programmed with OpenCL are even more complex and usually tailored to specific problems. Nevertheless, users may benefit from excellent speedups of naturally parallel applications, and from low power consumption which results from massive parallelization with moderate system frequency.

What about near future? How will develop high performance computers and the corresponding programming tools in the near future? Currently, the only possibility to increase computing power is to *increase parallelism* in algorithms and programs, and to *increase the number of cooperating processors*, which are often supported by massively parallel accelerators. Why is that so? The reason is that state-of-the-art production technology is already faced with **physical limits** dictated by space (e.g., dimension of transistors) and time (e.g., system frequency) [8].

Current high performance computers, containing millions of cores, can execute more than 10^{17} floating point operations per second (100 petaFLOPS). According to the Moore's law, the next challenge is to reach the *exascale barrier in the next decade*. However, due to above mentioned physical and technological limitations the validity of Moore's law is questionable. So it seems that the most effective approach to future parallel computing is an interplay of controlflow and dataflow paradigms, that is, in the **heterogeneous computing**. But programming of heterogeneous computers is still a challenging interdisciplinary task.

In this book, we did not describe programming of such extremely high performance computers; rather, we described and trained the reader for programming of parallel computers *at hand*, e.g., our personal computers, computers in cloud, or in computing clusters. Fortunately, the approaches and methodology of parallel programming is fairly independent of the complexity of computers.

In summary, it looks like that we cannot expect any significant shift in computing performance until a **new production technology** for computing devices is invented. Untill then the maximal exploitation of parallelism will be our delightful challenge.

Appendix A
Hints for making your computer a parallel machine

Practical advises for the installation of required supporting software for parallel program execution on different operating systems are given. Note, that this guide and internet links can change in the future, therefore always look for the up-to-date solution proposed by software tools providers.

A.1 Linux

OpenMP

OpenMP 4.5 has been a part of GNU GCC C/C++, the standard C/C++ compiler on Linux, by default since GCC's version 6 and thus it comes preinstalled on virtually any recent mainstream Linux distribution. You can check the version of your GCC C/C++ compiler by running command

```
$ gcc --version
```

The first line, e.g., something like

```
gcc (Ubuntu 6.3.0-12ubuntu2) 6.3.0 20170406
```

contains the information about the version of GCC C/C++ compiler (6.3.0 in this example).

Utility `time` can be used to measure the execution time of a given program. Furthermore, Gnome's System Monitor or the command-line utilities `top` (with separate-cpu-states displayed — press 1 once `top` starts) and `htop` can be used to monitor the load on individual logical cores.

MPI

The Message-Passing Interface (MPI) standard implementation can be already provided as a part of the operating system, most often as MPICH [1] or Open MPI [2, 10]. If it is not, it can usually be installed through the provided package management systems, for example, the apt in Ubuntu:

```
sudo apt install libmpich-dev
```

The MPICH is an open high-performance and widely portable implementation of the MPI, which is well maintained and supports the latest standards of the MPI. MPICH runs on parallel systems of all sizes, from multicore nodes to computer clusters in large supercomputers. Alternative open source implementations exists, e.g. Open MPI, with similar performances and user interface. Other implementations are dedicated to a specific hardware, some of them are commercial, however, the beauty of the MPI remains, your program will possibly executes with all of the MPI implementations, eventually after some initial difficulties. In the following, we will mostly use the acronym MPI, regardless of the actual implementation of the standard, except in cases if such a distinguishing is necessary.

Invoke the MPI execution manager: `>mpiexec` to check for the installed implementation of the MPI library on your computer. Either a note that the program is currently not installed or a help text will be printed. Let us assume that an Open MPI library is installed on your computer. The command: `>mpiexec -h` shows all the available options. Just a few of them will suffice for testing your programs.

We start working with typing a first program. Make your local directory, e.g. with OpenMPI:

```
>mkdir OMPI
```

retype the ”Hello World” program from Section 4.3 in your editor and save your code in file `OMPIHello.c`. Compile and link the program with a set-up for maximal speed:

```
>mpicc -O3 -o OMPIHello OMPIHello.c
```

which, beside compiling, also links appropriate MPI libraries with your program. Note, that on some system an additional option `-lm` could be needed for correct inclusion of all required files and libraries.

The compiled executable can be run by:

```
>mpiexec -n 3 OMPIHello
```

The output of the program should be in three lines, each line with a notice from a separate process:

```
Hello world from process 0 of 3
Hello world from process 1 of 3
Hello world from process 2 of 3
```

as the program has run on three processes, because the option `-n 3` was used. Note, that the line order is arbitrary, because there is no rule about the MPI process execution order. This issue is addressed in more details in Chapter 4.

OpenCL

First of all you need to download the newest drivers to your graphics card. This is important because OpenCL will not work if you don't have drivers that support OpenCL. To install OpenCL you need to download an implementation of OpenCL. The major graphic vendors NVIDIA, AMD and Intel have both released implementations of OpenCL for their GPUs. Besides the drivers, you should get the OpenCL headers and libraries included in the OpenCL SDK from your favourite vendor. The installation steps differ for each SDK and the OS you are running. Follow the installation manual of the SDK carefully. For OpenCL headers and libraries the main options you can choose from are: *NVIDIA CUDA Toolkit*, *AMD APP SDK* or *Intel SDK for OpenCL*. After the installation of drivers and SDK, you should the OpenCL headers:

```
#include<CL/cl.h>
```

If the OpenCL header and library files are located in their proper folders, the following command will compile an OpenCL program:

```
gcc prog.c -o prog -l OpenCL
```

A.2 macOS

OpenMP

Unfortunately the LLVM C/C++ compiler on macOS comes without OpenMP support (and the command `gcc` is simply a link to the LLVM compiler). To check your C/C++ compiler, run

```
$ gcc --version
```

If the output contains the line

```
Apple LLVM version 9.1.0 (clang-902.0.39.1)
```

where some numbers might change from one version to another, the compiler most likely do not support OpenMP. To use OpenMP, you have to install the original GNU

GCC C/C++ compiler (use MacPorts or Homebrew, for instance) which prints out something like

```
gcc-mp-7 (MacPorts gcc7 7.3.0_0) 7.3.0
```

informing that this is indeed the GNU GCC C/C++ compiler (version 7.3.0 in this example).

The running time can be measured in the same way as on Linux (see above). Monitoring the load on individual cores can be performed using macOS's Activity Monitor (open its CPU Usage window) or `htop` (but not with macOS's `top`).

MPI

In order to use MPI on macOS systems, XDeveloper and GNU compiler must be installed. Download XCode from the Mac App Store and install it by double-clicking the downloaded .dmg file. Use the command: `>mpiexec` to check for installed implementation of the MPI library on your computer. Either a note that the program is currently not installed or a help text will be printed.

If the latest stable release of Open MPI is not present, download it, for example, from the Open Source High Performance Computing website: `https://www.open-mpi.org/`. To install Open MPI on your computer, first extract the downloaded archive by typing the following command in your terminal (assuming that the latest stable release in 3.0.1):

```
>tar -zxvf openmpi-3.0.1.tar.gz
```

Then, prepare the `config.log` file needed for the installation. The `config.log` file collects information about your system:

```
>cd openmpi-3.0.1
>./configure --prefix=/usr/local
```

Finally, make the executables for installation and finalize the installation:

```
>make all
>sudo make install
```

After successful installation of Open MPI, we start working by typing our first program. Make your local directory, e.g. with: `>mkdir OMPI`.

Copy or retype the "Hello World" program from Section 4.3 in your editor and save your code in file `OMPIHello.c`.

Compile and link the program with:

```
>mpicc -O3 -o OMPIHello OMPIHello.c
```

Execute your program "Hello World" with:

```
>mpiexec  -n 3 OMPIHello.
```

The output of the program should be similar to the output of the ”Hello World” MPI program from Appendix A.1.

OpenCL

If you are using Apple Mac OS X, the Apple's OpenCL implementation should already be installed on your system. MAC OS X 10.6 and later ships with a native implementation of OpenCL. The implementation consists of the OpenCL application programming interface, the OpenCL runtime engine and the OpenCL compiler.

OpenCL is fully supported by Xcode. If you use Xcode, all you need to do is to include the OpenCL header file:

```
#include <OpenCL/opencl.h>
```

A.3 MS Windows

OpenMP

There are several options for using OpenMP on Microsoft Windows. To follow the examples in the book as closely as possible, it is best to use Linux Subsystem for Windows 10. If a Linux distribution brings recent enough version of GNU GCC C/C++ compiler, e.g., Debian, one can compile OpenMP programs with it. Furthermore, one can use commands `time`, `top` and `htop` to measure and monitor programs.

Another option is of course using Microsoft Visual C++ compiler. OpenMP has been supported by it since 2005. Apart from using it form within Microsoft Visual Studio, one can start `x64 Native Tools Command Prompt for VS 2017` where programs can be compiled and run as follows:

```
> cl /openmp /O2 hello-world.c
> set OMP_NUM_THREADS=8
> hello-world.exe
```

With PowerShell run in `x64 Native Tools Command Prompt for VS 2017`, programs can be compiled and run as

```
> powershell
> cl /openmp /O2 fibonacci.c
> $env:OMP_NUM_THREADS=8
> ./fibonacci.exe
```

Within PowerShell, the running time of a program can be measured using the command `Measure-Command` as follows:

```
> Measure-Command {./hello-world.exe}
```

Regardless of the compiler used, the execution of the programs can be monitored using Task Manager (open the CPU tab within the Resource Monitor).

MPI

More detailed instructions for installation of necessary software for compiling and running the Microsoft MPI can be found, for example, on: `https://blogs.technet.microsoft.com/windowshpc/2015/02/02/how-to-compile-and-run-a-simple-ms-mpi-program/`. A short summary is listed below:

- Download stand-alone redistributables for Microsoft SDK `msmpisdk.msi` and Microsoft MPI `MSMpiSetup.exe` installers from: `https://www.microsoft.com/en-us/download/confirmation.aspx?id=55991`, which will provide execute utility for MPI programs `mpiexec.exe` and MPI service âĂŞ process manager `smpd.exe`.
- Set the MS-MPI environment variables in a terminal window by:
  `C:\Windows\System32>set MSMPI`, which should print the following lines, if the installation of SDK and MSMPI has been correctly completed:

  ```
  MSMPI_BIN=C:\Program Files\Microsoft MPI\Bin\
  MSMPI_INC=C:\Program Files (x86)\Microsoft SDKs\MPI\Include\
  MSMPI_LIB32=C:\Program Files (x86)\Microsoft SDKs\MPI\Lib\x86\
  MSMPI_LIB64=C:\Program Files (x86)\Microsoft SDKs\MPI\Lib\x64\
  ```

  The command: `>mpiexec` should respond with basic library options.
- Download Visual Studio Community C++ 2017 from: `https://www.visualstudio.com/vs/visual-studio-express/` and install the compiler on your computer, e.g. by selecting a simple desktop development installation.
- You will be forced to restart your computer. After a restarting, start Visual Studio and create `File/New/Project/Windows Console Application`, named e.g. MSMPIHello, with default settings except:

  1. To include the proper header files, open `Project Property pages` and insert in `C/C++/General` under `Additional Include Directories`:
     `$(MSMPI_INC);$(MSMPI_INC)\x64`
     if 64 bit solution will be build. Use `..\x86` for 32 bits.
  2. To setup the linker library in Project Property pages insert in `Linker/General` under `Additional Library Directories`:
     `$(MSMPI_LIB64)`
     if 64 bit platform will be used. Use `$(MSMPI_LIB32)` for 32 bits.
  3. In `Linker/Input` under `Additional Dependencies` add:
     `msmpi.lib;`

4. Close the Project Property window and check in the main Visual Studio window that `Release` solution configuration is selected and select also a solution platform of your computer, e.g. x64.

- Copy or retype "Hello World" program from Section 4.3 and build the project.
- Open a Command prompt window, change directory to the folder where the project was built, e.g. `..\source\repos\MSMPIHello\x64\Debug` and run the program from the command window with execute utility:

  ```
  mpiexec -n 3 MSMPIHello
  ```

  that should result in the same output as in Appendix A.1, with three lines, each with a notice from a separate process.

OpenCL

First of all you need to download the newest drivers to your graphics card. This is important because OpenCL will not work if you don't have drivers that support OpenCL. To install OpenCL you need to download an implementation of OpenCL. The major graphic vendors NVIDIA, AMD and Intel have both released implementations of OpenCL for their GPUs. Besides the drivers, you should get the OpenCL headers and libraries included in the OpenCL SDK from your favourite vendor. The installation steps differ for each SDK and the OS you are running. Follow the installation manual of the SDK carefully. For OpenCL headers and libraries the main options you can choose from are: *NVIDIA CUDA Toolkit*, *AMD APP SDK* or *Intel SDK for OpenCL*. After the installation of drivers and SDK, you should the OpenCL headers:

```
#include<CL/cl.h>
```

Suppose you are using Visual Studio 2013, you need to tell the compiler where the OpenCL headers are located and tell the linker where to find the OpenCL .lib files.

References

1. MPICH: High-Performance Portable MPI. https://www.mpich.org/. Accessed 28 Dec 2017
2. Open MPI: A High Performance Message Passing Library. https://www.open-mpi.org/. Accessed 28 Dec 2017
3. Atallah, M., Blanton, M. (eds.): Algorithms and Theory of Computation Handbook, chap. 25. Chapman and Hall (2010)
4. Chandra, R., Menon, R., Dagum, L., Kohr, D., Maydan, D., McDonald, J.: Parallel Programming in OpenMP. Morgan Kaufmann (2000)
5. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, 3rd edn. The MIT Press (2009)
6. Dally, W.J., Towles, B.: Principles and Practices of Interconnection Networks. Morgan Kaufmann (2004)
7. Duato, J., Yalamanchili, S., Ni, L.: Interconnection Networks. Morgan Kaufmann (2002)
8. Flynn, M.J., Mencer, O., Milutinovic, V., Rakocevic, G., Stenstrom, P., Trobec, R., Valero, M.: Moving from petaflops to petadata. Commun. ACM **56**(5), 39–42 (2013). DOI 10.1145/2447976.2447989. URL `http://doi.acm.org/10.1145/2447976.2447989`
9. Foster, I.: Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1995)
10. Gabriel, E., Fagg, G.E., Bosilca, G., Angskun, T., Dongarra, J.J., Squyres, J.M., Sahay, V., Kambadur, P., Barrett, B., Lumsdaine, A., Castain, R.H., Daniel, D.J., Graham, R.L., Woodall, T.S.: Open MPI: Goals, concept, and design of a next generation MPI implementation. In: Proceedings, 11th European PVM/MPI Users' Group Meeting, pp. 97–104. Budapest, Hungary (2004)
11. Gaster, B., Howes, L., Kaeli, D.R., Mistry, P., Schaa, D.: Heterogeneous Computing with OpenCL, 1st edn. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2011)
12. Grama, A., Gupta, A., Karypis, V., Kumar, V.: Introduction to Parallel Computing, 2nd edn. Pearson (2003)
13. Hopcroft, J.E., Ullman, J.D.: Introduction to Automata Theory, Languages, and Computation. Addison-Wesley, Reading, MA, USA (1979)
14. Jason Long: Hands On OpenCL: An open source two-day lecture course for teaching and learning OpenCL. https://handsonopencl.github.io/ (2018). Accessed 25 Jun 2018
15. Khronos Group: OpenCL: The open standard for parallel programming of heterogeneous systems. https://www.khronos.org/opencl/f (2000–2018). Accessed 25 Jun 2018
16. MPI Forum: MPI: A message-passing interface standard (Version 3.1). Tech. rep., Knoxville, TN, USA (2015)
17. Munshi, A., Gaster, B., Mattson, T.G., Fung, J., Ginsburg, D.: OpenCL Programming Guide, 1st edn. Addison-Wesley Professional (2011)
18. OpenMP Architecture Review Board: OpenMP Application Programming Interface, Version 4.5, November 2015. http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf (1997–2015). Accessed 18 Dec 2017

19. OpenMP Architecture Review Board: OpenMP Application Programming Interface Examples, Version 4.5.0, November 2016. http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf (1997–2016). Accessed 18 Dec 2017
20. OpenMP Architecture Review Board: OpenMP. http://www.openmp.org (2012). Accessed 18 Dec 2017
21. van der Pas, R., Stotzer, E., Terboven, C.: Using OpenMP – The Next Step: Affinity, Accelerators, Tasking, and SIMD (Scientific and Engineering Computation). The MIT Press (2017)
22. Robič, B.: The Foundations of Computability Theory. Springer (2015)
23. Scarpino, M.: A Gentle Introduction to OpenCL. http://www.drdobbs.com/parallel/a-gentle-introduction-to-opencl/231002854. Accessed 10 Apr 2018
24. Scarpino, M.: OpenCL in Action: How to Accelerate Graphics and Computations. Manning Publications (2011). URL `http://amazon.com/o/ASIN/1617290173/`
25. Trobec, R.: Two-dimensional regular d-meshes. Parallel Comput. **26**(13-14), 1945–1953 (2002)
26. Trobec, R., Vasiljević, R., Tomašević, M., Milutinović, V., Beivide, R., Valero, M.: Interconnection networks in petascale computer systems: A survey. ACM Comput. Surv. **49**(3), 44:1–44:24 (2016)

Index