Chapter 1
Introduction

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

1.1 Classical Use of Parallelism

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

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

DOI 10.1007/978-3-642-04818-0_1, © Springer-Verlag Berlin Heidelberg 2010

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

1.2 Parallelism in Today's Hardware

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

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

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

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

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

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

1.3 Basic Concepts

A first step in parallel programming is the design of a parallel algorithm or program for a given application problem. The design starts with the decomposition of the computations of an application into several parts, called **tasks**, which can be computed in parallel on the cores or processors of the parallel hardware. The decomposition into tasks can be complicated and laborious, since there are usually many different possibilities of decomposition for the same application algorithm. The size of tasks (e.g., in terms of the number of instructions) is called **granularity** and there is typically the possibility of choosing tasks of different sizes. Defining the tasks of an application appropriately is one of the main intellectual works in the development of a parallel program and is difficult to automate. **Potential parallelism** is an inherent property of an application algorithm and influences how an application can be split into tasks.

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

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

Specific **barrier operations** offer another form of coordination which is available for both shared memory and distributed memory machines. All processes or threads have to wait at a barrier synchronization point until all other processes or threads have also reached that point. Only after all processes or threads have executed the code before the barrier, they can continue their work with the subsequent code after the barrier.

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

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

1.4 Overview of the Book

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

Chapter 3 considers popular parallel programming models and paradigms and discusses how the inherent parallelism of algorithms can be presented to a parallel runtime environment to enable an efficient parallel execution. An important part of this chapter is the description of mechanisms for the coordination of parallel programs, including synchronization and communication operations. Moreover, mechanisms for exchanging information and data between computing resources for different memory models are described. Chapter 4 is devoted to the performance analysis of parallel programs. It introduces popular performance or cost measures that are also used for sequential programs, as well as performance measures that have been developed for parallel programs. Especially, popular communication patterns for distributed address space architectures are considered and their efficient implementations for specific interconnection networks are given.

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

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

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