1 Why Do We Need Parallel Programming

Chapter Summary
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

R. Trobec et al., *Introduction to Parallel Computing*,
Undergraduate Topics in Computer Science,
https://doi.org/10.1007/978-3-319-98833-7_1

allowed even higher execution speeds of programs, regardless of the processor and memory frequency.
- Second, any commercial computer, tablet, and smartphone contain 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 **several multi-core 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 *classicaln-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 Einstein's 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*, 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 abovementioned 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 $n = 640{,}000$ to $n = 1{,}000{,}000$ 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 first 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*.