CHAPTER 1

Introduction

1.1 OVERVIEW

Although computer performance has evolved exponentially in the past, there have always been applications that demand more processing power than a single state-of-the-art processor can provide. To respond to this demand, multiple processing units are employed conjointly to collaborate on the execution of one application. Computer systems that consist of multiple processing units are referred to as *parallel systems*. Their aim is to speed up the execution of an application through the collaboration of the processing units. With the introduction of dual-core and multicore processors by IBM, AMD, Intel, and others, even mainstream PCs have become parallel systems.

Even though the area of parallel computing has existed for many decades, programming a parallel system for the execution of a single application is still a challenging problem, profoundly more challenging than programming a single processor, or sequential, system. Figure 1.1 illustrates the process of parallel programming. Apart from the formulation of the application in a programming language—this is the programming for sequential systems—the application must be divided into subtasks to allow the distribution of the application's computational load among the processors. Generally, there are dependences between the tasks that impose a partial order on their execution. Adhering to this order is essential for the correct execution of the application. A crucial step of parallel programming is the allocation of the tasks to the processors and the definition of their execution order. This step, which is referred to as *scheduling*, fundamentally determines the efficiency of the application's parallelization, that is, the speedup of its execution in comparison to a single processor system.

The complexity of parallel programming motivates research into automatic parallelization techniques and tools. One particularly difficult part of automatic parallelization is the scheduling of the tasks onto the processors. Basically, one can distinguish between *dynamic* and *static scheduling*. In dynamic scheduling, the decision as to which processor executes a task and when is controlled by the runtime system. This is mostly practical for independent tasks. In contrast, static scheduling

Task Scheduling for Parallel Systems, by Oliver Sinnen

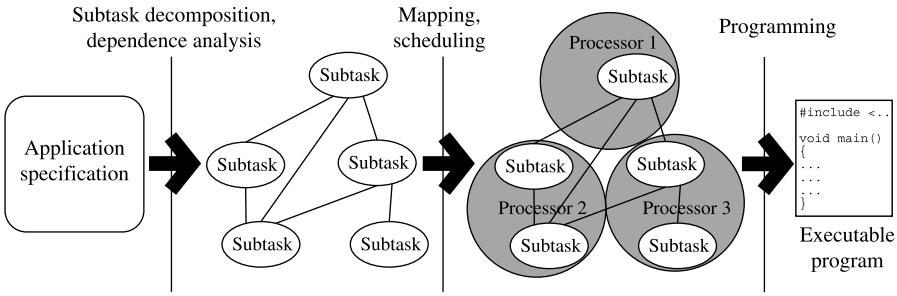

Figure 1.1. Parallel programming—process of parallelization.

means that the processor allocation, often called mapping, and the ordering of the tasks are determined at compile time. The advantage of static scheduling is that it can include the dependences and communications among the tasks in its scheduling decisions. Furthermore, since the scheduling is done at compile time, the execution is not burdened with the scheduling overhead.

In its general form (i.e., without any restrictions on the application's type or structure), static scheduling is often referred to as *task scheduling*. The applications, or parts of them, considered in task scheduling can have arbitrary task and dependence structures. They are represented as directed acyclic graphs (DAGs), called *task graphs*, where a node reflects a task and a directed edge a communication between the incident nodes. Weights associated with the nodes and edges represent the computation and communication costs, respectively. For example, consider the small program segment in Figure 1.2 and its corresponding task graph. Each line of the program is represented by one node and the edges reflect the communications among the nodes; for instance, line 2 reads the data line 1 has written into the variable *a*, hence the edge from node 1 to node 2.

The task graph represents the task and communication structure of the program, which is determined during the subtask decomposition and the dependence analysis. An edge imposes a precedence constraint between the incident nodes: the origin node must be executed before the destination node. For example, in Figure 1.2 node 1 must

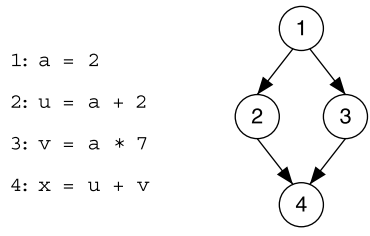

Figure 1.2. Example of task graph representing a small program segment.

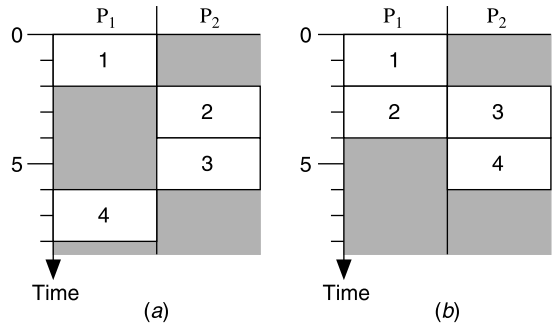

Figure 1.3. Two sample schedules of the task graph in Figure 1.2.

be executed before nodes 2 and 3 (both read the value of a written in line 1), which in turn must be executed before node 4 (line 4 adds the results of line 2 (u) and line 3 (v)).

The challenge of task scheduling is to find a spatial and temporal assignment of the nodes onto the processors of the target system, which results in the fastest possible execution, while respecting the precedence constraints expressed by the edges. As an example, consider the two schedules in Figure 1.3 of the above task graph on two processors, P_1 and P_2. For simplicity, it is here assumed that the nodes have identical weights of two time units, and all edge weights are zero. In both schedules each processor executes two nodes, yet schedule (b) is shorter than schedule (a). The reason is the precedence constraints among the nodes: in the schedule (a), the two nodes that can be executed in parallel, nodes 2 and 3, are allocated to the same processor. In schedule (b), they are allocated to different processors and executed concurrently.

What is a trivial problem in this example becomes very difficult with larger, more complex task graphs. In fact, finding a schedule of minimal length for a given task graph is, in its general form, an NP-hard problem (Ullman [192]); that is, its associated decision problem is NP-complete and an optimal solution cannot be found in polynomial time (unless NP = P). As a consequence of the NP-hardness of scheduling, an entire area emerged that deals with all aspects of task scheduling, ranging from its theoretical analysis to heuristics and approximation techniques that produce near optimal solutions.

This book is devoted to this area of task scheduling for parallel systems. Through a thorough introduction to parallel systems, their architecture, and parallel programming, task scheduling is first carefully set into the context of the parallelization process. The program representation model of task scheduling—the task graph—is studied in detail and compared with other graph-based models. This is one of the first attempts to analyze and compare the major graph models for the representation of programs.

After this ground-laying introduction, the task scheduling problem is formally defined and its theoretical background is rigorously analyzed. Throughout the entire book, this unifying theoretical framework is employed, making the study of task

scheduling very consistent. For example, task scheduling without the consideration of communication costs is studied as a special case of the general problem that recognizes the costs of communication.

But the effort of having a comprehensive and easy to understand treatment of task scheduling does not stop there. After establishing the theory, the focus is on common concepts and techniques encountered in many task scheduling algorithms, rather than presenting a loose survey of algorithms. Foremost, these are the two fundamental scheduling heuristics—*list scheduling* and *clustering*—which are studied and analyzed in great detail. The book continues by looking at more advanced topics of task scheduling, namely, *node insertion*, *node duplication*, and *genetic algorithms*.

While the concepts and techniques are extracted and treated separately, the framework is backed up with references to many proposed algorithms. This approach has several advantages: (1) common concepts and terminology simplify the understanding, analysis, and comparison of algorithms; (2) it is easier to evaluate the impact of a technique when it is detached from other techniques; and (3) the design of new algorithms may be inspired to use new combinations of the presented techniques.

This book also explores further aspects of the theoretical background of scheduling. One aspect is scheduling on heterogeneous processors, including the corresponding scheduling model and the adapted algorithms. Another aspect is the study of variations of the general task scheduling problem. A comprehensive survey of these variations, which again can be treated as special cases of the general problem, shows that most of them are also NP-hard problems.

The book then goes beyond the classic approach to task scheduling by studying scheduling under other, more accurate, parallel system models. Classic scheduling is based on the premise that the target system consists of a set of fully connected processors, which means each processor has a direct communication link to every other processor. Interprocessor communication is performed by a dedicated communication subsystem, in a way that is completely free of contention. It follows that an unlimited number of interprocessor communications can be realized concurrently without the involvement of the processors. Qualitative analysis and recent experimental evaluations show that not all of these assumptions are fulfilled for the majority of parallel systems. This issue is addressed in two steps.

In the first step, a model is investigated that extends task scheduling toward contention awareness. Following the spirit of the unifying scheduling framework, the investigated contention model is a general and unifying model in terms of network representation and contention awareness. It allows modeling of arbitrary heterogeneous systems, relating to processors and communication links, and integrates the awareness of end-point and network contention. Adapting scheduling algorithms to the contention model is straightforward. Exemplarily, it is studied how list scheduling can be employed under the contention model.

In the second step, the scheduling framework is extended further to integrate involvement of the processors in communication. The resulting model inherits all abilities of the contention model and allows different types of processor involvement in communication. Processor involvement has a relatively strong impact on the scheduling process and therefore demands new approaches. Several approaches

to handle this difficulty are analyzed and the adaptation of scheduling heuristics is discussed.

Throughout this book, numerous figures and examples illustrate the discussed concepts. Exercises at the end of each chapter deepen readers' understanding.

1.2 ORGANIZATION

This book is organized as follows.

Chapter 2 reviews the relevant background of parallel computing, divided into two parts. The first part discusses parallel computers, their architectures and their communication networks. The second part returns to parallel programming and the parallelization process, reviewing subtask decomposition and dependence analysis in detail.

Chapter 3 provides a profound analysis of the three major graph models for the representation of computer programs: dependence graph, flow graph, and task graph. It starts with the necessary concepts of graph theory and then formulates a common principle for graph models representing computer programs. While the focus is on the task graph, the broad approach of this chapter is crucial in order to establish a comprehensive understanding of the task graph, its principle, its relations to other models, and its motivations and limitations.

Chapter 4 is devoted to the fundamentals of task scheduling. It carefully introduces terminology, basic definitions, and the target system model. The scheduling problem is formulated and subsequently the NP-completeness of the associated decision problem is proved. One of the aims of this chapter is to provide the reader with a unifying and consistent conceptual framework. Consequently, task scheduling without communication costs is studied as a special case of the general problem. Again, the complexity is discussed, including the NP-completeness of this problem. The chapter then returns to the task graph model to analyze its properties in connection with task scheduling.

Chapter 5 addresses the two fundamental heuristics for scheduling—list scheduling and clustering. Both are discussed in general terms, following the expressed intention of this book to focus on common concepts and techniques. For list scheduling, a distinction is made between static and dynamic node priorities. Given a processor allocation, list scheduling can also be employed to construct a schedule. The area of clustering can be broken down into a few conceptually different approaches. Those are analyzed, followed by a discussion on how to go from clustering to scheduling.

Chapter 6 has a look at more advanced aspects of task scheduling. The first two sections deal with node insertion and node duplication. Both techniques can be employed in many scheduling heuristics. Again, for the sake of a better understanding, they are studied detached from such heuristics. The chapter then returns to more theoretical aspects of task scheduling. Integrating heterogeneous processors into scheduling can be done quite easily. A survey of variants of the general scheduling problem shows that scheduling remains NP-hard in most cases even after restricting the problem.

The last aspects to be considered in this chapter are genetic algorithms and how they can be applied to the scheduling problem.

Chapter 7 investigates how to handle contention for communication resources in task scheduling. The chapter begins with an overview of existing contention aware scheduling algorithms, followed by an outline of the approach taken in this book. Next, an enhanced topology graph is introduced, based on a thorough analysis of communication networks and routing. Contention awareness is achieved with edge scheduling, which is investigated in the third section. The next section shows how task scheduling is made contention aware by integrating edge scheduling and the topology graph into the scheduling process. Adapting algorithms for scheduling under the contention model is analyzed in the last section, with the focus on list scheduling.

Chapter 8 investigates processor involvement in communication and its integration into task scheduling. It begins by classifying interprocessor communication into three types and by analyzing their main characteristics. To integrate processor involvement into contention scheduling, the scheduling model is adapted. The new model implies changes to the existing scheduling techniques. General approaches to scheduling under the new model are investigated. Using these approaches, two scheduling heuristics are discussed for scheduling under the new model, namely, list scheduling and two-phase heuristics.