TASK SCHEDULING FOR PARALLEL SYSTEMS

Oliver Sinnen
Department of Electrical and Computer Engineering
The University of Aukland
New Zealand

WILEY-INTERSCIENCE
A JOHN WILEY & SONS, INC., PUBLICATION

TASK SCHEDULING FOR PARALLEL SYSTEMS

THE WILEY BICENTENNIAL–KNOWLEDGE FOR GENERATIONS

Each generation has its unique needs and aspirations. When Charles Wiley first opened his small printing shop in lower Manhattan in 1807, it was a generation of boundless potential searching for an identity. And we were there, helping to define a new American literary tradition. Over half a century later, in the midst of the Second Industrial Revolution, it was a generation focused on building the future. Once again, we were there, supplying the critical scientific, technical, and engineering knowledge that helped frame the world. Throughout the 20th Century, and into the new millennium, nations began to reach out beyond their own borders and a new international community was born. Wiley was there, expanding its operations around the world to enable a global exchange of ideas, opinions, and know-how.

For 200 years, Wiley has been an integral part of each generation's journey, enabling the flow of information and understanding necessary to meet their needs and fulfill their aspirations. Today, bold new technologies are changing the way we live and learn. Wiley will be there, providing you the must-have knowledge you need to imagine new worlds, new possibilities, and new opportunities.

Generations come and go, but you can always count on Wiley to provide you the knowledge you need, when and where you need it!

William J. Pesce

WILLIAM J. PESCE
PRESIDENT AND CHIEF EXECUTIVE OFFICER

Peter Booth Wiley

PETER BOOTH WILEY
CHAIRMAN OF THE BOARD

TASK SCHEDULING FOR PARALLEL SYSTEMS

Oliver Sinnen
Department of Electrical and Computer Engineering
The University of Aukland
New Zealand

WILEY-INTERSCIENCE
A JOHN WILEY & SONS, INC., PUBLICATION

Published by John Wiley & Sons, Inc., Hoboken, New Jersey.
Published simultaneously in Canada.

Library of Congress Cataloging-in-Publication Data:

Sinnen, Oliver, 1971-
Task scheduling for parallel systems / by Oliver Sinnen.
p. cm.
Includes bibliographical references and index.
978-0-471-73576-2
1. Parallel processing (Electronic computers) 2. Multitasking (Computer science) 3. Computer scheduling. I. Title.
QA76.58.S572 2007
044′.35—dc22 2006052157

Printed in the United States of America.

10 9 8 7 6 5 4 3 2 1

à Patrícia

CONTENTS

PREFACE

Even though the area of parallel computing has existed for many decades, programming a parallel system is still a challenging problem, much more challenging than programming a single processor system. With the current dual-core and multicore processors from IBM, AMD, Intel, and others, mainstream PCs have entered the realm of parallel systems. The investigation and understanding of the foundations of parallel computing is therefore more important than ever.

One of these foundations is task scheduling. To execute a program consisting of several tasks on a parallel system, the tasks must be arranged in space and time on the multiple processors. In other words, the tasks must be mapped to the processors and ordered for execution. This so-called task scheduling is a very complex problem and crucially determines the efficiency of the parallel system. In fact, task scheduling is an NP-hard problem; that is, an optimal solution generally cannot be found in polynomial time (unless P = NP). This has been motivating the development of many heuristics for its near optimal solution.

This book is devoted to task scheduling for parallel systems. Anyone who gets involved for the first time in task scheduling is overwhelmed by the enormous wealth of heuristics, models, and methods that have been contributed during the last decades. One of my main objectives for this book is to bring order into this jungle of task scheduling. However, the book does not simply categorize and order scheduling heuristics. Instead, it investigates and presents task scheduling by extracting and discussing common models, methods, and techniques, and by setting them into relation. Hence, this book is not a mere survey of scheduling algorithms, but rather an attempt at a consistent and unifying theoretical framework.

Another objective I have with this book is to go beyond the classic approach to task scheduling by studying scheduling under more advanced and accurate system models. These system models consider heterogeneity, contention for communication resources, and involvement of the processor in communication. For efficient and accurate task scheduling, a realistic system model is most crucial. This book is the first publication that discusses advanced system models for task scheduling in a comprehensive form.

Task Scheduling for Parallel Systems is targeted at practicing professionals, researchers, and students. For those who are new to task scheduling, the first chapters carefully introduce parallel systems and their programming, setting task scheduling into the context of the program parallelization process. Practitioners involved in parallel programming will gain an understanding of fundamental aspects of the parallelization process. This knowledge will help them to write more efficient code.

Compiler and parallelization tool developers will benefit from a deeper understanding of the scheduling problem, which is also a generalization of many other problems they face (e.g., loop scheduling). A chapter on graph models promotes the understanding of these relations. For task scheduling researchers, this book serves as a comprehensive reference, based on a unifying framework. The research community will especially value the later chapters on advanced scheduling and sophisticated scheduling models. Graduate students of parallel computing and compiler courses can use this book to thoroughly study task scheduling, which is supported by the exercises at the end of each chapter. The extensive index and the large number of bibliographic references make this book a valuable tool for everybody interested in task scheduling.

For a brief introduction to task scheduling and an overview of this book, including a short summary of each chapter, refer to Chapter 1.

OLIVER SINNEN
The University of Aukland
New Zealand

ACKNOWLEDGMENTS

This book has its roots in my PhD thesis, which I presented at the Instituto Superior Técnico, Technical University of Lisbon in Portugal. I therefore want to sincerely thank Prof. Leonel Sousa again for his academic advice, the fruitful discussions, and his criticism during that time. In this context I also thank Prof. Frode Sandnes, Oslo University College, Norway, who introduced me to genetic algorithms and has been an invaluable research colleague for many years.

I would like to thank The University of Auckland for the opportunity to write this book while being a lecturer in the Department of Electrical and Computer Engineering.

Cover design: Digital image of a painting created by Patrícia Raquel de Vasconcelos da Silva (2006).

OLIVER SINNEN
The University of Aukland
New Zealand

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

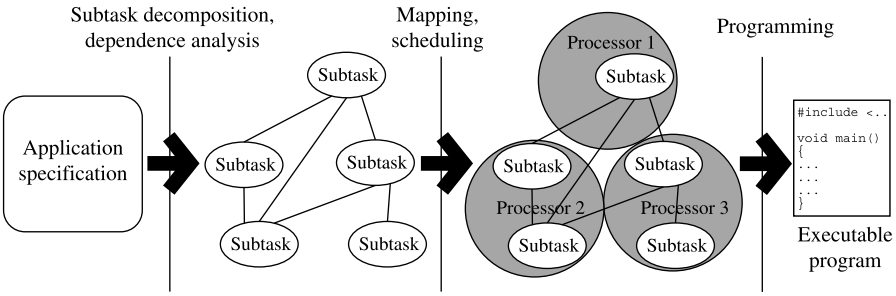

Figure 1.1. Parallel programming—process of parallelization.

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

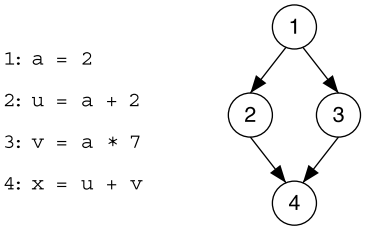

Figure 1.2. Example of task graph representing a small program segment.

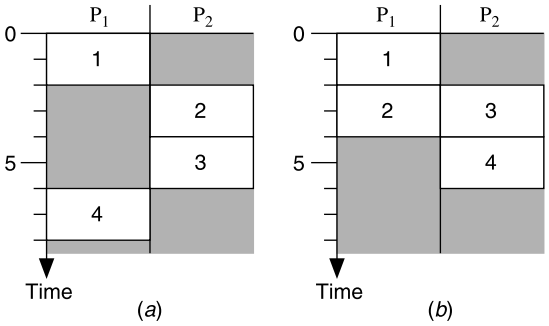

Figure 1.3. Two sample schedules of the task graph in Figure 1.2.

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.

CHAPTER 2

Parallel Systems and Programming

This chapter reviews parallel systems and their programming. The intention is to establish the necessary background and terminology for the following chapters. It begins with the basis of parallel computing—parallel systems—and discusses their architectures and communication networks. In this context, it also addresses programming models for parallel systems. The second part of the chapter is devoted to the parallelization process of parallel programming. A general overview presents the three components of the parallelization process: subtask decomposition, dependence analysis, and scheduling. The subsequent sections discuss subtask decomposition and dependence analysis, which build the foundation for task scheduling.

2.1 PARALLEL ARCHITECTURES

Informally, a parallel computer can be characterized as a system where multiple processing elements cooperate in executing one or more tasks. This is in contrast to the von Neumann model of a sequential computer, where only one processor executes the task. The numerous existing parallel architectures and their different approaches require some kind of classification.

2.1.1 Flynn's Taxonomy

In a frequently referenced article by Flynn [67], the design of a computer is characterized by the flow (or stream) of instructions and the flow (or stream) of data. The taxonomy classifies according to the multiplicity of the instruction and the data flows. The resulting four possible combinations are shown in Table 2.1.

The SISD (single instruction single data) architecture corresponds to the conventional sequential computer. One instruction is executed at a time on one data item. Although the combination MISD (multiple instruction single data) does not seem to be meaningful, pipeline architectures, as found in all modern processors, can be considered MISD (Cosnard and Trystram [45]).

Task Scheduling for Parallel Systems, by Oliver Sinnen

Table 2.1. Flynn's Taxonomy

	Single Data	Multiple Data
Single Instruction	SISD	SIMD
Multiple Instruction	MISD	MIMD

In SIMD (single instruction multiple data) architectures, which are also called data parallel or vector architectures, multiple processing elements (PEs) execute the same instruction on different data items. Figure 2.1(*a*) shows the SIMD structure with one central control unit and multiple processing elements. The central control unit issues the same instruction stream to each PE, which works on its own data set. Especially for regular computations from the area of science and engineering (e.g., signal processing), where computations can be expressed as vector and matrix operations, the SIMD architecture is well adapted.

There are only a few examples of systems that have a pure SIMD architecture, for instance, the early vector machines (e.g., the Cray-1 or the Hitachi S-3600) (van der Steen and Dongarra [193]). Today, the SIMD architecture is often encountered within a vector processor, that is, one chip consisting of the central control unit together with multiple processing elements. A parallel system can be built from multiple vector processors and examples for such systems are given later. Also, most of today's mainstream processor architectures feature an SIMD processing unit, for example, the MMX/SSE unit in the Intel Pentium processor line or the AltiVec unit in the PowerPC processor architecture.

The second parallel architecture in the taxonomy has MIMD (multiple instruction multiple data) streams, depicted in Figure 2.1(*b*). In contrast to the SIMD structure, every PE has its own control unit (CU). Therefore, the processor elements operate

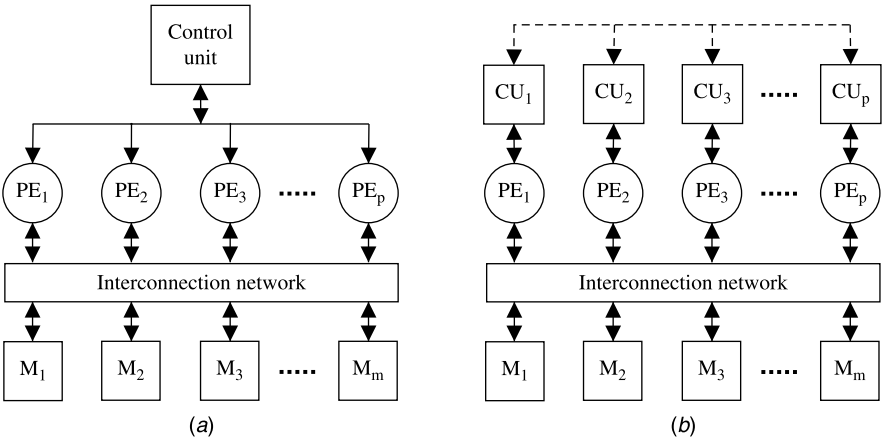

Figure 2.1. SIMD (*a*) and MIMD (*b*) architecture.

independently of each other and execute independent instructions on different data streams. A parallel execution of a global task (i.e., the collaboration of the processing elements) is achieved through synchronization and data exchange between the PEs via the interconnection network. Examples for MIMD architectures are given in the following discussion of memory architectures.

An MIMD architecture can simulate an SIMD architecture by executing the same program on all the processors, which is called SPMD (single program multiple data) mode. In general, however, executing the same program on all processors is not the same as executing the same instruction stream, since processors might execute different parts of the same program depending on their processor identification numbers. The term *processor* stands here for the combination of control unit plus processing element. From now on, this definition of a processor shall be used, if not otherwise stated.

As mentioned earlier, modern parallel systems often consist of multiple vector processors, for example, the Cray J90 or the NEC SX-6 (van der Steen and Dongarra [193]). The Earth Simulator, a NEC SX-6 based system, was the world's fastest computer in 2002–2004 [186]. Within Flynn's taxonomy, these systems can be considered to have an MIMD architecture with an SIMD subarchitecture.

2.1.2 Memory Architectures

It is generally agreed that not all aspects of parallel architectures are taken into account by Flynn's taxonomy. For both the design and the programming model of a parallel system, the memory organization is a very important issue not considered by that classification.

The memory organization of a parallel system can be divided into two aspects: the location and the access policy of the memory. Regarding the location, memory is either *centralized* or *distributed* with the processors. For both cases, systems with a common memory, distributed or not, to which all processors have full access, are called *shared-memory* machines. In systems where there is no such shared memory, processors have to use explicit means of communication like *message passing*. With these two aspects of memory organization in mind, the three most common memory organizations can be examined.

Centralized Memory In a centralized memory multiprocessor, illustrated in Figure 2.2, memory is organized as a central resource for all processors. This typically results in a *uniform memory access* (UMA) characteristic, where the access time to any memory location is identical for every processor. Since the common memory can be accessed by all processors, these systems are called *centralized shared-memory multiprocessors* (Hennessy and Patterson [88]). Due to the UMA characteristic, systems with this architecture are also often called symmetric multiprocessors (SMP).

Distributed Memory The alternative to a centralized architecture is an architecture where the memory is physically distributed with the processors. These systems can be further distinguished according to their memory access policy.

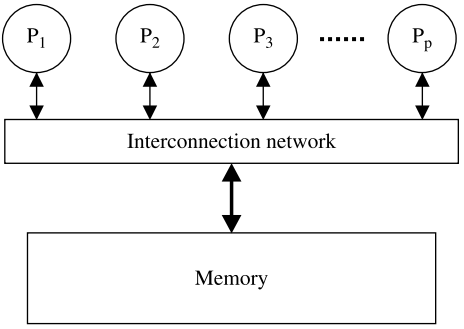

Figure 2.2. Centralized memory multiprocessor.

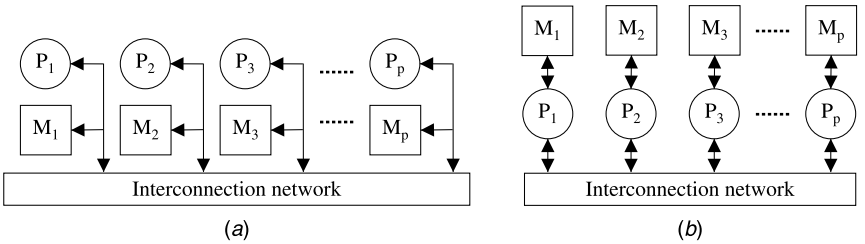

Figure 2.3. Distributed memory multiprocessors: (*a*) shared-memory and (*b*) message passing (memory access goes through processors).

A *distributed shared-memory multiprocessor* (Hennessy and Patterson [88]), as illustrated in Figure 2.3(*a*), integrates the distributed memories into a global address space. Every processor has full access to the memory; however, in general with a *nonuniform memory access* (NUMA) characteristic, as reading from or writing to, local memory is faster than from and to remote memory.

Systems without shared memory are called *distributed memory multiprocessors* or, according to the way the processors communicate, *message passing architectures*[1] (Culler and Singh [48]). Figure 2.3(*b*) displays a distributed memory multiprocessor without shared memory. The difference to the distributed shared-memory multiprocessor is that the local memories are only accessible through the respective processors.

Memory Hierarchy Some distributed memory systems, especially large systems with shared memory, often use some kind of hierarchy for the memory organization. A common example is that a small number of processors (2–4) share one central memory—the processors and the memory comprise a computing node—and

[1]To distinguish the two distributed memory architectures, the supplements shared-memory or message passing shall be used. Otherwise both types are meant.

multiples of these nodes are interconnected on a higher level. Examples of systems with such a hierarchical memory architecture are the Sequent NUMA-Q, the SGI Origin 2000/3000 series, and the IBM Blue Gene/L (van der Steen and Dongarra [193]), which is the world's fastest computer at the time this is written [186].

Motivations The following paragraphs briefly look at the motivations for the different architectures.

Centralized shared-memory architectures are an intuitive extension of a single processor architecture; however, the contention for the communication resources to the central memory significantly limits the scalability of these machines. Bus-based systems therefore have a small number of processors (usually ≤ 8), for example, systems with $\times 86$ processors, and only more sophisticated interconnection networks allow these systems to scale up to 64 processors. For example, in the Sun Enterprise series (van der Steen and Dongarra [193]), systems with low model numbers (e.g., 3000) are connected by a bus, whereas the high-end model, the Sun Enterprise 10000, uses a crossbar (see Section 2.2) for up to 64 processors.

In contrast, distributed memory architectures with message passing allow a much simpler system design, but their programming becomes more complicated. In fact, commodity PCs can be connected via a commodity network (e.g., Ethernet) to build a so-called cluster of workstations or PCs (Patterson [147], Sterling et al. [181]). The big advantage of distributed memory systems is their much better scalability. Hence, it is no surprise that the massively parallel processors (MPPs) are distributed memory systems using message passing with up to hundred thousand processors (e.g., Thinking Machines CM-5, Intel Paragon, and IBM Blue Gene/L).

Distributed shared-memory architectures try to integrate both approaches. They provide the ease of the shared-memory programming paradigm and benefit from the scalability of distributed memory systems, for instance, Cray T3D/T3E, the SGI Origin 2000/3000 series, or the HP SuperDome series (van der Steen and Dongarra [193]). Yet, shared-memory programming of these architectures can have limited efficiency in as much as the heterogeneous access times to memory are often hidden from the programmer.

2.1.3 Programming Paradigms and Models

Shared-Memory Versus Message Passing Programming The programming paradigms for parallel systems have a strong correspondence to the memory access policies of multiprocessors. Fundamentally, one can distinguish between shared-memory and message passing programming. In the former paradigm, every processor has full access to the shared memory, and communication between the parallel processors is done implicitly via the memory. Only concurrent access to the same memory location needs explicit synchronization of the processors. In message passing programming, every exchange of data among processors must be explicitly expressed with send and receive commands.

It must be noted that the employed programming paradigm does not always correspond to the underlying memory organization of the target system. Message passing

can be utilized on both shared-memory and message passing architectures. In a shared-memory system, the passing of a message is often implemented as a simple memory copy. Even distributed shared memory can be emulated on message passing machines with an additional software layer (e.g., survey by Protić et al. [154]).

Parallel Random Access Machine (PRAM) PRAM (Fortune and Wyllie [68]) is a popular machine model for algorithm design and complexity analysis. Essentially, the simple model assumes an ideal centralized shared-memory machine with synchronously working processors. PRAMs can be further classified according to how one memory cell can be accessed: only exclusively by one processor or concurrently by various processors. Memory access to different cells by different processors can always be performed concurrently.

The advantage of PRAM is its simplicity and its similarity to the sequential von Neumann model. Yet, owing to the increasing gap between processing and communication speed, it has become more and more unrealistic.

LogP With the proposal of the LogP model, Culler et al. [46, 47] recognized the fact that the widely used PRAM model is unrealistic due to its assumption of cost-free interprocessor communication, especially for distributed systems. The LogP model gained its name from the parameters that are used to describe a parallel system:

- L: Upper bound on the *latency*, or delay, incurred in communicating a message from a source to a destination processor.
- o: *Overhead*—time during which a processor is engaged in sending or receiving a message; during this time the processor cannot perform other operations.
- g: *Gap*—minimal time between consecutive message transmissions or between consecutive message receptions; the reciprocal of g corresponds to the per-processor bandwidth.
- P: Number of processors.

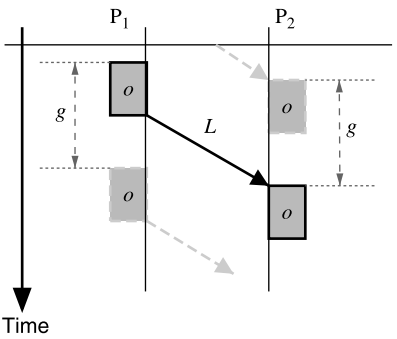

Figure 2.4. Interprocessor communication in LogP.

Furthermore, the structure of the processor network is not described by LogP, but its capacity is limited to $\lceil L/g \rceil$ simultaneous message transfers between all processors. An implicit parameter is the message size M, which consists of one or a small number of words. Based on this parameter, an interprocessor message transfer in the LogP model proceeds as illustrated in Figure 2.4. In contrast to PRAM, LogP is an asynchronous model.

2.2 COMMUNICATION NETWORKS

Fast communication is crucial for an efficient parallel system. A determining aspect of the communication behavior is the network and its topology. In the previous section, some kind of interconnection network for communication among the units of the parallel system was supposed. This section reviews the principal network types, of which each offers a different trade-off between cost and performance.

Initially, interconnection networks can be classified into *static* and *dynamic networks*. Static networks have fixed connections between the units of the system with point-to-point communication links. In a dynamic network, the connections between units of the parallel system are established dynamically through switches when requested. Based on this difference, static, and dynamic networks are sometimes referred to as *direct* and *indirect* networks, respectively (Grama et al. [82], Quinn [156]).

2.2.1 Static Networks

The essential characteristic of a static network is its topology, as the interconnections between the units of the parallel system are fixed. Most static networks are processor networks used in distributed memory systems, where every processor has its own local memory.

Processor network topologies are usually represented as undirected graphs[2]: a vertex represents a processor, together with its local memory and a switch, and an undirected edge represents a communication link between two processors (Cosnard and Trystram [45], Culler and Singh [48], Grama et al. [82]). Figure 2.5(b) depicts an example for a network graph consisting of four processors. Figure 2.5(a) illustrates the implicit association of memory and a switch with each processor.

Once a topology is modeled as an undirected graph, terminology from graph theory can be utilized for its characterization. The *degree* of a vertex is defined as the number of its incident edges, denoted by δ. The eccentricity of a vertex is the largest distance, in terms of the number of edges, from that vertex to any other vertex. Furthermore, the *diameter* of an undirected graph, denoted by D, is defined as the maximum eccentricity of all vertices of the graph. Another notable indicator for a network is its *bisection width*. It is defined as the minimum number of edges that have to be removed to

[2]Basic graph concepts are introduced in Section 3.1 and the undirected graph model of topologies will be defined more formally in Section 7.1. For the current purpose, this informal definition suffices.

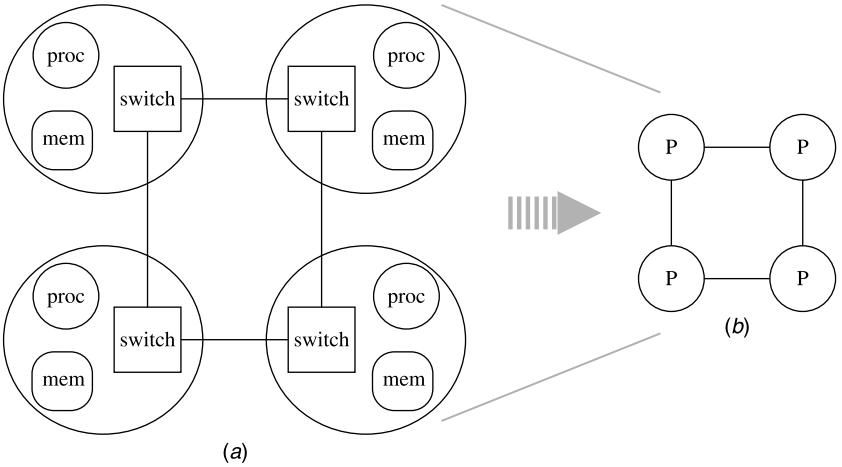

Figure 2.5. An undirected graph representing a processor network (*b*); (*a*) an illustration that the switch and the memory associated with each processor are implicit in the common representation of processor networks.

partition the network into two equal halves. In other words, it is the number of edges that cross a cut of a network into two equal halves.

To achieve a network with a small maximum communication time, the goal is to have a small diameter D. At the same time, the mean degree $\bar{\delta}$ of the network should be small, since it determines the hardware costs. Last but not least, the network should have a large bisection width, because it lowers the contention for the communication links. However, the bisection width is also a measure for the network costs, as it provides a lower bound on the area or volume of its packaging (Grama et al. [82]).

Fully Connected Networks A network in which every processor has a direct link to any other processor, as depicted in Figure 2.6, is called fully connected. It has

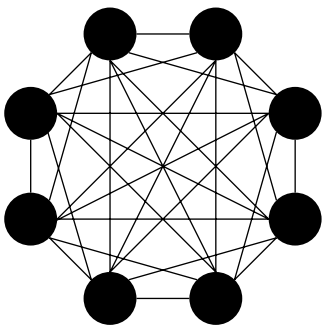

Figure 2.6. An 8-processor fully connected network.