25

Parallel Algorithms

Guy E. Blelloch
Carnegie Mellon University

Bruce M. Maggs
Duke University and Akamai Technologies

25.1 Introduction

The subject of this chapter is the design and analysis of parallel algorithms. Most of today's algorithms are sequential, that is, they specify a sequence of steps in which each step consists of a single operation. These algorithms are well suited to today's computers, which basically perform operations in a sequential fashion. Although the speed at which sequential computers operate has been improving at an exponential rate for many years, the improvement is now coming at greater and greater cost. As a consequence, researchers have sought more cost-effective improvements by building "parallel" computers—computers that perform multiple operations in a single step. In order to solve a problem efficiently on a parallel computer, it is usually necessary to design an algorithm that specifies multiple operations on each step, i.e., a parallel algorithm.

As an example, consider the problem of computing the sum of a sequence A of n numbers. The standard algorithm computes the sum by making a single pass through the sequence, keeping

a running sum of the numbers seen so far. It is not difficult however, to devise an algorithm for computing the sum that performs many operations in parallel. For example, suppose that, in parallel, each element of A with an even index is paired and summed with the next element of A, which has an odd index, i.e., $A[0]$ is paired with $A[1]$, $A[2]$ with $A[3]$, and so on. The result is a new sequence of $\lceil n/2 \rceil$ numbers that sum to the same value as the sum that we wish to compute. This pairing and summing step can be repeated until, after $\lceil \log_2 n \rceil$ steps, a sequence consisting of a single value is produced, and this value is equal to the final sum.

The parallelism in an algorithm can yield improved performance on many different kinds of computers. For example, on a parallel computer, the operations in a parallel algorithm can be performed simultaneously by different processors. Furthermore, even on a single-processor computer the parallelism in an algorithm can be exploited by using multiple functional units, pipelined functional units, or pipelined memory systems. Thus, it is important to make a distinction between the parallelism in an algorithm and the ability of any particular computer to perform multiple operations in parallel. Of course, in order for a parallel algorithm to run efficiently on any type of computer, the algorithm must contain at least as much parallelism as the computer, for otherwise resources would be left idle. Unfortunately, the converse does not always hold: some parallel computers cannot efficiently execute all algorithms, even if the algorithms contain a great deal of parallelism. Experience has shown that it is more difficult to build a general-purpose parallel computer than a general-purpose sequential computer.

The remainder of this chapter consists of nine sections. We begin in Section 25.2 with a discussion of how to model parallel computers. Next, in Section 25.3 we cover some general techniques that have proven useful in the design of parallel algorithms. Sections 25.4 through 25.8 present algorithms for solving problems from different domains. We conclude in Section 25.9 with a discussion of current research topics, a collection of defining terms, and finally sources for further information.

Throughout this chapter, we assume that the reader has some familiarity with sequential algorithms and asymptotic notation and analysis.

25.2 Modeling Parallel Computations

The designer of a sequential algorithm typically formulates the algorithm using an abstract model of computation called the random-access machine (RAM) model [2], Chapter 1. In this model, the machine consists of a single processor connected to a memory system. Each basic CPU operation, including arithmetic operations, logical operations, and memory accesses, requires one time step. The designer's goal is to develop an algorithm with modest time and memory requirements. The RAM model allows the algorithm designer to ignore many of the details of the computer on which the algorithm will ultimately be executed, but captures enough detail that the designer can predict with reasonable accuracy how the algorithm will perform.

Modeling parallel computations is more complicated than modeling sequential computations because in practice parallel computers tend to vary more in organization than do sequential computers. As a consequence, a large portion of the research on parallel algorithms has gone into the question of modeling, and many debates have raged over what the "right" model is, or about how practical various models are. Although there has been no consensus on the right model, this research has yielded a better understanding of the relationship between the models. Any discussion of parallel algorithms requires some understanding of the various models and the relationships among them.

In this chapter we divide parallel models into two classes: **multiprocessor models** and **work-depth models**. In the remainder of this section we discuss these two classes and how they are related.

25.2.1 Multiprocessor Models

A multiprocessor model is a generalization of the sequential RAM model in which there is more than one processor. Multiprocessor models can be classified into three basic types: local-memory machine models, modular memory machine models, and parallel random-access machine (PRAM) models. Figure 25.1 illustrates the structure of these machine models. A local-memory machine model consists of a set of n processors each with its own local-memory. These processors are attached to a common communication network. A modular memory machine model consists of m memory modules and n processors all attached to a common network. An n-processor PRAM model consists of a set of n processors all connected to a common shared memory [32,37,38,77].

The three types of multiprocessors differ in the way that memory can be accessed. In a local-memory machine model, each processor can access its own local memory directly, but can access the memory in another processor only by sending a memory request through the network. As in the RAM model, all local operations, including local memory accesses, take unit time. The time taken to access the memory in another processor, however, will depend on both the capabilities of the communication network and the pattern of memory accesses made by other processors, since these other accesses could congest the network. In a modular memory machine model, a processor accesses the memory in a memory module by sending a memory request through the network. Typically the processors and memory modules are arranged so that the time for any processor to access any memory module is roughly uniform. As in a local-memory machine model, the exact amount of

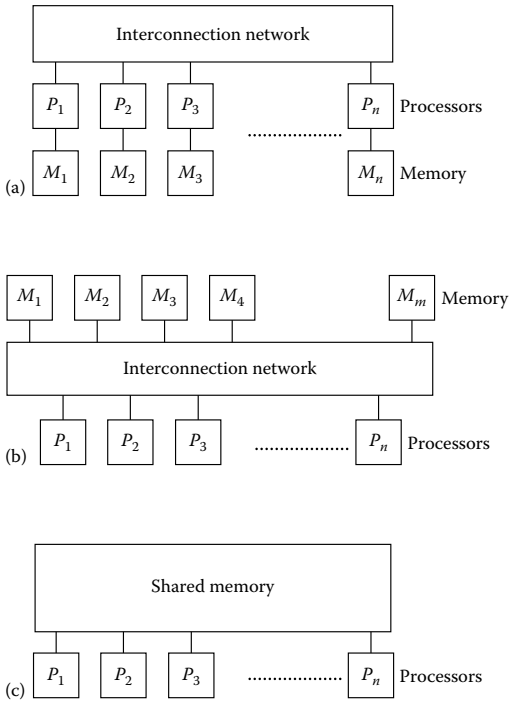

FIGURE 25.1 The three types of multiprocessor machine models: (a) a local-memory machine model; (b) a modular memory machine model; and (c) a PRAM model.

time depends on the communication network and the memory access pattern. In a **PRAM model**, a processor can access any word of memory in a single step. Furthermore, these accesses can occur in parallel, i.e., in a single step, every processor can access the shared memory.

The PRAM models are controversial because no real machine lives up to its ideal of unit-time access to shared memory. It is worth noting, however, that the ultimate purpose of an abstract model is not to directly model a real machine, but to help the algorithm designer produce efficient algorithms. Thus, if an algorithm designed for a PRAM model (or any other model) can be translated to an algorithm that runs efficiently on a real computer, then the model has succeeded. In Section 25.2.4 we show how an algorithm designed for one parallel machine model can be translated so that it executes efficiently on another model.

The three types of multiprocessor models that we have defined are broad and allow for many variations. The local-memory machine models and modular memory machine models may differ according to their network topologies. Furthermore, in all three types of models, there may be differences in the operations that the processors and networks are allowed to perform. In the remainder of this section we discuss some of the possibilities.

25.2.1.1 Network Topology

A network is a collection of switches connected by communication channels. A processor or memory module has one or more communication ports that are connected to these switches by communication channels. The pattern of interconnection of the switches is called the network topology. The topology of a network has a large influence on the performance and also on the cost and difficulty of constructing the network. Figure 25.2 illustrates several different topologies.

The simplest network topology is a bus. This network can be used in both local-memory machine models and modular memory machine models. In either case, all processors and memory modules are typically connected to a single bus. In each step, at most one piece of data can be written onto the bus. This data might be a request from a processor to read or write a memory value, or it might be the response from the processor or memory module that holds the value. In practice, the advantage of using a bus is that it is simple to build and, because all processors and memory modules can observe

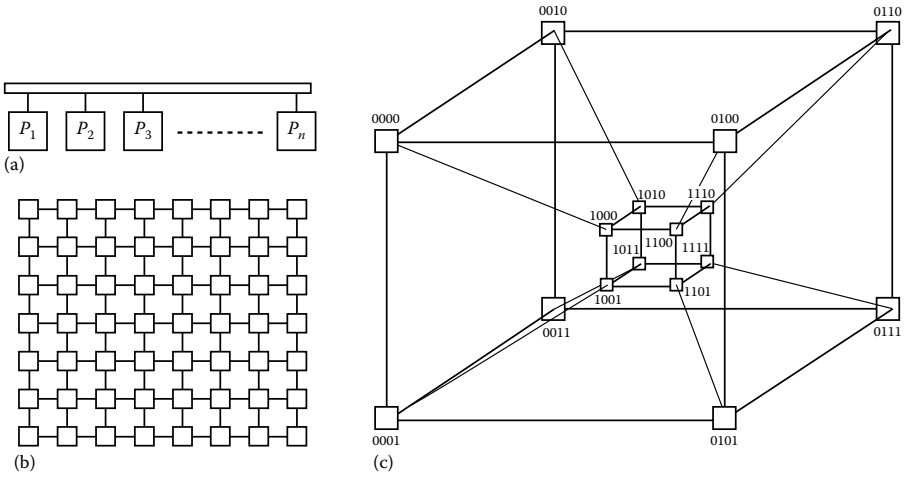

FIGURE 25.2 (a) Bus, (b) two-dimensional mesh, and (c) hypercube network topologies.

the traffic on the bus, it is relatively easy to develop protocols that allow processors to cache memory values locally. The disadvantage of using a bus is that the processors have to take turns accessing the bus. Hence, as more processors are added to a bus, the average time to perform a memory access grows proportionately.

A two-dimensional mesh is a network that can be laid out in a rectangular fashion. Each switch in a mesh has a distinct label (x, y) where $0 \leq x \leq X - 1$ and $0 \leq y \leq Y - 1$. The values X and Y determine the length of the sides of the mesh. The number of switches in a mesh is thus $X \cdot Y$. Every switch, except those on the sides of the mesh, is connected to four neighbors: one to the north, one to the south, one to the east, and one to the west. Thus, a switch labeled (x, y), where $0 < x < X - 1$ and $0 < y < Y - 1$, is connected to switches $(x, y + 1)$, $(x, y - 1)$, $(x + 1, y)$, and $(x - 1, y)$. This network typically appears in a local-memory machine model, i.e., a processor along with its local memory is connected to each switch, and remote memory accesses are made by routing messages through the mesh. Figure 25.2b shows an example of an 8×8 mesh.

Several variations on meshes are also popular, including three-dimensional meshes, toruses, and hypercubes. A torus is a mesh in which the switches on the sides have connections to the switches on the opposite sides. Thus, every switch (x, y) is connected to four other switches: $(x, y + 1 \bmod Y)$, $(x, y - 1 \bmod Y)$, $(x + 1 \bmod X, y)$, and $(x - 1 \bmod X, y)$. A hypercube is a network with 2^n switches in which each switch has a distinct n-bit label. Two switches are connected by a communication channel in a hypercube if and only if the labels of the switches differ in precisely one bit position. A hypercube with 16 switches is shown in Figure 25.2c.

A multistage network is used to connect one set of switches called the input switches to another set called the output switches through a sequence of stages of switches. Such networks were originally designed for telephone networks [15]. The stages of a multistage network are numbered 1 through L, where L is the depth of the network. The switches on stage 1 are the input switches, and those on stage L are the output switches. In most multistage networks, it is possible to send a message from any input switch to any output switch along a path that traverses the stages of the network in order from 1 to L. Multistage networks are frequently used in modular memory computers; typically processors are attached to input switches, and memory modules are attached to output switches. A processor accesses a word of memory by injecting a memory access request message into the network. This message then travels through the network to the appropriate memory module. If the request is to read a word of memory, then the memory module sends the data back through the network to the requesting processor. There are many different multistage network topologies. Figure 25.3a, for example, shows a depth-2 network that connects 4 processors to 16 memory modules. Each switch in this network has two channels at the bottom and four channels at the top. The ratio of processors to memory modules in this example is chosen to reflect the fact that, in practice, a processor is capable of generating memory access requests faster than a memory module is capable of servicing them.

A fat-tree is a network structured like a tree [56]. Each edge of the tree, however, may represent many communication channels, and each node may represent many network switches (hence the name "fat"). Figure 25.3b shows a fat-tree with the overall structure of a binary tree. Typically the

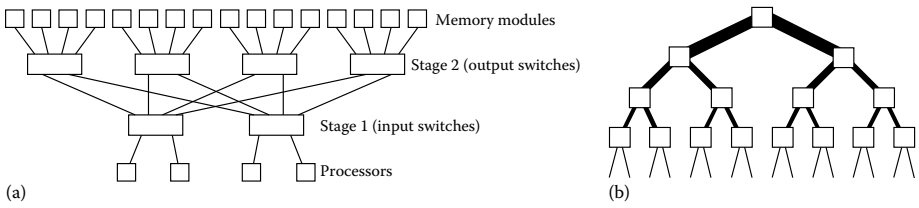

FIGURE 25.3 (a) 2-level multistage network and (b) fat-tree network topologies.

DOCKET ALARM

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS
Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS
Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS
Sync your system to PACER to automate legal marketing.