PERFORMANCE ANALYSIS FORMULAS

Amdahl's Law

Let f be the fraction of operations in a computation that must be performed sequentially, where $0 \leq f \leq 1$. The maximum speedup ψ achievable by a parallel computer with p processors performing the computation is

$$\psi \leq \frac{1}{f + (1 - f)/p}$$

Gustafson-Barsis's Law

Given a parallel program solving a problem of size n using p processors, let s denote the fraction of total execution time spent in serial code. The maximum speedup ψ achievable by this program is

$$\psi \leq p + (1 - p)s$$

Karp-Flatt Metric

Given a parallel computation exhibiting speedup ψ on p processors, where $p > 1$, the experimentally determined serial fraction e is defined to be

$$e = \frac{1/\psi - 1/p}{1 - 1/p}$$

Isoefficiency Relation

Suppose a parallel system exhibits efficiency $\varepsilon(n, p)$, where n denotes problem size and p denotes number of processors. Define $C = \varepsilon(n, p)/(1 - \varepsilon(n, p))$. Let $T(n, 1)$ denote sequential execution time, and let $T_o(n, p)$ denote parallel overhead (total amount of time spent by all processors performing communications and redundant computations). In order to maintain the same level of efficiency as the number of processors increases, problem size must be increased so that the following inequality is satisfied:

$$T(n, 1) \geq CT_o(n, p)$$

Parallel Programming
in C with MPI and OpenMP

Michael J. Quinn
Oregon State University

Higher Education

Boston Burr Ridge, IL Dubuque, IA Madison, WI New York San Francisco St. Louis
Bangkok Bogotá Caracas Kuala Lumpur Lisbon London Madrid Mexico City
Milan Montreal New Delhi Santiago Seoul Singapore Sydney Taipei Toronto

PARALLEL PROGRAMMING IN C WITH MPI AND OPENMP
International Edition 2003

10 09 08 07 06
20 09 08 07
CTF SLP

Library of Congress Cataloging-in-Publication Data

Quinn, Michael J. (Michael Jay)
Parallel programming in C with MPI and OpenMP / Michael J. Quinn.—1st ed.
p. cm.
ISBN 007-282256-2
1. C (Computer program language). 2. Parallel programming (Computer science). I.Title.
QA76.73.C15Q55 2004
005.13'3—dc21 2003046371
CIP

When ordering this title, use ISBN 007-123265-6

Printed in Singapore

www.mhhe.com

With gratitude for their love, support, and guidance,
I dedicate this book to my parents,
Edward and Georgia Quinn.

Anibal Meneses Vivros.

BRIEF TABLE OF CONTENTS

CONTENTS

PREFACE

This book is a practical introduction to parallel programming in C using the MPI (Message Passing Interface) library and the OpenMP application programming interface. It is targeted to upper-division undergraduate students, beginning graduate students, and computer professionals learning this material on their own. It assumes the reader has a good background in C programming and has had an introductory class in the analysis of algorithms.

Fortran programmers interested in parallel programming can also benefit from this text. While the examples in the book are in C, the underlying concepts of parallel programming with MPI and OpenMP are essentially the same for both C and Fortran programmers.

In the past twenty years I have taught parallel programming to hundreds of undergraduate and graduate students. In the process I have learned a great deal about the sorts of problems people encounter when they begin "thinking in parallel" and writing parallel programs. Students benefit from seeing programs designed and implemented step by step. My philosophy is to introduce new functionality "just in time." As much as possible, every new concept appears in the context of solving a design, implementation, or analysis problem. When you see the symbol

in a page margin, you'll know I'm presenting a key concept.

The first two chapters explain when and why parallel computing began and gives a high-level overview of parallel architectures. Chapter 3 presents Foster's parallel algorithm design methodology and shows how it is used through several case studies. Chapters 4, 5, 6, 8, and 9 demonstrate how to use the design methodology to develop MPI programs that solve a series of progressively more difficult programming problems. The 27 MPI functions presented in these chapters are a robust enough subset to implement parallel programs for a wide variety of applications. These chapters also introduce functions that simplify matrix and vector I/O. The source code for this I/O library appears in Appendix B.

The programs of Chapters 4, 5, 6, and 8 have been benchmarked on a commodity cluster of microprocessors, and these results appear in the text. Because new generations of microprocessors appear much faster than books can be produced, readers will observe that the processors are several generations old. The point of presenting the results is not to amaze the reader with the speed of the computations. Rather, the purpose of the benchmarking is to demonstrate that knowledge of the latency and bandwidth of the interconnection network, combined with information about the performance of a sequential program, are often sufficient to allow reasonably accurate predictions of the performance of a parallel program.

Chapter 7 focuses on four metrics for analyzing and predicting the performance of parallel systems: Amdahl's Law, Gustafson-Barsis' Law, the Karp-Flatt metric, and the isoefficiency metric.

Chapters 10–16 provide additional examples of how to analyze a problem and design a good parallel algorithm to solve it. At this point the development of MPI programs implementing the parallel algorithms is left to the reader. I present Monte Carlo methods and the challenges associated with parallel random number generation. Later chapters present a variety of key algorithms: matrix multiplication, Gaussian elimination, the conjugate gradient method, finite difference methods, sorting, the fast Fourier transform, backtrack search, branch-and-bound search, and alpha-beta search.

Chapters 17 and 18 are an introduction to the new shared-memory programming standard OpenMP. I present the features of OpenMP as needed to convert sequential code segments into parallel ones. I use two case studies to demonstrate the process of transforming MPI programs into hybrid MPI/OpenMP programs that can exhibit higher performance on multiprocessor clusters than programs based solely on MPI.

This book has more than enough material for a one-semester course in parallel programming. While parallel programming is more demanding than typical programming, it is also more rewarding. Even with a teacher's instruction and support, most students are unnerved at the prospect of harnessing multiple processors to perform a single task. However, this fear is transformed into a feeling of genuine accomplishment when they see their debugged programs run much faster than "ordinary" C programs. For this reason, programming assignments should play a central role in the course.

Fortunately, parallel computers are more accessible than ever. If a commercial parallel computer is not available, it is a straightforward task to build a small cluster out of a few PCs, networking equipment, and free software.

Figure P.1 illustrates the precedence relations among the chapters. A solid arrow from A to B indicates chapter B depends heavily upon material presented in chapter A. A dashed arrow from A to B indicates a weak dependence. If you cover the chapters in numerical order, you will satisfy all of these precedences. However, if you would like your students to start programming in C with MPI as quickly as possible, you may wish to skip Chapter 2 or only cover one or two sections of it. If you wish to focus on numerical algorithms, you may wish to skip Chapter 5 and introduce students to the function `MPI_Bcast` in another way. If you would like to start by having your students programming Monte Carlo algorithms, you can jump to Chapter 10 immediately after Chapter 4. If you want to cover OpenMP before MPI, you can jump to Chapter 17 after Chapter 3.

I thank everyone at McGraw-Hill who helped me create this book, especially Betsy Jones, Michelle Flomenhoft, and Kay Brimeyer. Thank you for your sponsorship, encouragement, and assistance. I also appreciate the help provided by Maggie Murphy and the rest of the compositors at Interactive Composition Corporation.

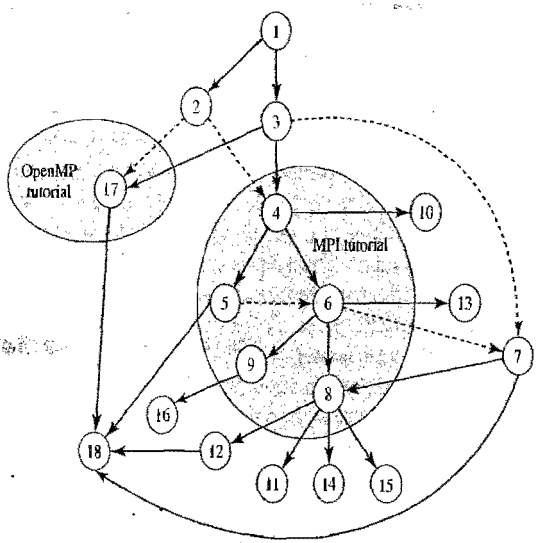


Figure P.1 Dependences among the chapters. A solid arrow indicates a strong dependence; a dashed arrow indicates a weak dependence.

I am indebted to the reviewers who carefully read the manuscript, correcting errors, pointing out weak spots, and suggesting additional topics. My thanks to: A. P. W. Bohm, Colorado State University; Thomas Cormen, Dartmouth College; Narsingh Deo, University of Central Florida; Philip J. Hatcher, University of New Hampshire; Nickolas S. Jovanovic, University of Arkansas at Little Rock; Dinesh Mehta, Colorado School of Mines; Zina Ben Miled, Indiana University-Purdue University, Indianapolis; Paul E. Plassman, Pennsylvania State University; Quinn O. Snell, Brigham Young University; Ashok Srinivasan, Florida State University; Xian-He Sun, Illinois Institute of Technology; Virgil Wallentine, Kansas State University; Bob Weems, University of Texas at Arlington; Kay Zemoudel, California State University-San Bernardino; and Jun Zhang, University of Kentucky.

Many people at Oregon State University also lent me a hand. Rubin Landau and Henri Jansen helped me understand Monte Carlo algorithms and the detailed balance condition, respectively. Students Charles Sauerbier and Bernd Michael Kelm suggested questions that made their way into the text. Tim Budd showed me how to incorporate PostScript figures into LaTeX documents. Jalal Haddad provided technical support. Thank you for your help!

Finally, I am grateful to my wife, Victoria, for encouraging me to get back into textbook writing. Thanks for the inspiring Christmas present: *Chicken Soup for the Writer's Soul: Stories to Open the Heart and Rekindle the Spirit of Writers.*

Michael J. Quinn
Corvallis, Oregon

CHAPTER

Motivation and History

Well done is quickly done.
Caesar Augustus

1.1 INTRODUCTION

Are you one of those people for whom "fast" isn't fast enough? Today's workstations are about a hundred times faster than those made just a decade ago, but some computational scientists and engineers need even more speed. They make great simplifications to the problems they are solving and still must wait hours, days, or even weeks for their programs to finish running.

Faster computers let you tackle larger computations. Suppose you can afford to wait overnight for your program to produce a result. If your program suddenly ran 10 times faster, previously out-of-reach computations would now be within your grasp. You could produce in 15 hours an answer that previously required nearly a week to generate.

Of course, you *could* simply wait for CPUs to get faster. In about five years single CPUs will be 10 times faster than they are today (a consequence of Moore's Law). On the other hand, if you can afford to wait five years, you must not be in that much of a hurry! Parallel computing is a proven way to get higher performance *now*.

What's parallel computing?

Parallel computing is the use of a parallel computer to reduce the time needed to solve a single computational problem. Parallel computing is now considered a standard way for computational scientists and engineers to solve problems in areas as diverse as galactic evolution, climate modeling, aircraft design, and molecular dynamics.

What's a parallel computer?

A **parallel computer** is a multiple-processor computer system supporting parallel programming. Two important categories of parallel computers are multicomputers and centralized multiprocessors.

As its name implies, a **multicomputer** is a parallel computer constructed out of multiple computers and an interconnection network. The processors on different computers interact by passing messages to each other.

In contrast, a **centralized multiprocessor** (also called a **symmetrical multiprocessor** or **SMP**) is a more highly integrated system in which all CPUs share access to a single global memory. This shared memory supports communication and synchronization among processors.

We'll study centralized multiprocessors, multicomputers, and other parallel computer architectures in Chapter 2.

What's parallel programming?

Parallel programming is programming in a language that allows you to explicitly indicate how different portions of the computation may be executed concurrently by different processors. We'll discuss various kinds of parallel programming languages in more detail near the end of this chapter.

Is parallel programming really necessary?

A lot of research has been invested in the development of compiler technology that would allow ordinary Fortran 77 or C programs to be translated into codes that would execute with good efficiency on parallel computers with large numbers of processors. This is a very difficult problem, and while many experimental parallelizing[1] compilers have been developed, at the present time commercial systems are still in their infancy. The alternative is for you to write your own parallel programs.

Why should I program using MPI and OpenMP?

MPI (Message Passing Interface) is a standard specification for message-passing libraries. Libraries meeting the standard are available on virtually every parallel computer system. Free libraries are also available in case you want to run MPI on a network of workstations or a parallel computer built out of commodity components (PCs and switches). If you develop programs using MPI, you will be able to reuse them when you get access to a newer, faster parallel computer.

Increasingly, parallel computers are being constructed out of symmetrical multiprocessors. Within each SMP, the CPUs have a shared address space. While MPI is a perfectly satisfactory way for processors in different SMPs to communicate with each other, OpenMP is a better way for processors within a single SMP

[1] **parallelize** *verb*: to make parallel.

to interact. In Chapter 18 you'll see an example of how a hybrid MPI/OpenMP program can outperform an MPI-only program on a practical application.

By working through this book, you'll learn a little bit about parallel computer hardware and a lot about parallel program development strategies. That includes parallel algorithm design and analysis, program implementation and debugging, and ways to benchmark and optimize your programs.

1.2 MODERN SCIENTIFIC METHOD

Classical science is based on observation, theory, and physical experimentation. Observation of a phenomenon leads to a hypothesis. The scientist develops a theory to explain the phenomenon and designs an experiment to test that theory. Usually the results of the experiment require the scientist to refine the theory, if not completely reject it. Here, observation may again take center stage.

Classical science is characterized by physical experiments and models. For example, many physics students have explored the relationship between mass, force, and acceleration using paper tape, pucks, and air tables. Physical experiments allow scientists to test theories, such as Newton's first law of motion, against reality.

In contrast, contemporary science is characterized by observation, theory, experimentation, *and numerical simulation* (Figure 1.1). Numerical simulation is an increasingly important tool for scientists, who often cannot use physical experiments to test theories because they may be too expensive or time-consuming, because they may be unethical, or because they may be impossible to perform. The modern scientist compares the behavior of a numerical simulation, which

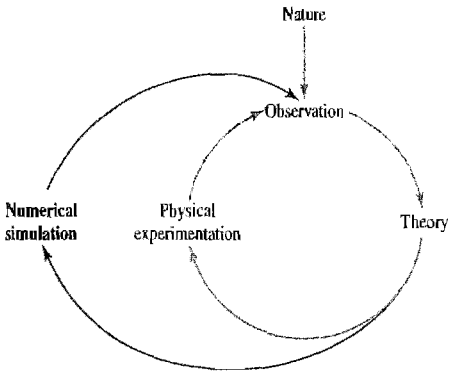


Figure 1.1 The introduction of numerical simulation distinguishes the contemporary scientific method from the classical scientific method.

implements the theory, to data collected from nature. The differences cause the scientist to revise the theory and/or make more observations.

Many important scientific problems are so complex that solving them via numerical simulation requires extraordinarily powerful computers. These complex problems, often called **grand challenges** for science, fall into several categories [73]:

1. Quantum chemistry, statistical mechanics, and relativistic physics
2. Cosmology and astrophysics
3. Computational fluid dynamics and turbulence
4. Materials design and superconductivity
5. Biology, pharmacology, genome sequencing, genetic engineering, protein folding, enzyme activity, and cell modeling
6. Medicine, and modeling of human organs and bones
7. Global weather and environmental modeling

While grand challenge problems emerged in the late 1980s as a stimulus for further developments in high-performance computing, you can view the entire history of electronic computing as the quest for higher performance.

1.3 EVOLUTION OF SUPERCOMPUTING

The United States government has played a key role in the development and use of high-performance computers. During World War II the U.S. Army paid for the construction of the ENIAC in order to speed the calculation of artillery tables. In the 30 years after World War II, the U.S. government used high-performance computers to design nuclear weapons, break codes, and perform other national security–related applications.

Supercomputers are the most powerful computers that can be built [60]. (As computer speeds increase, the bar for "supercomputer" status rises, too.) The term *supercomputer* first came into widespread use with the introduction of the Cray-1 supercomputer in 1976. The Cray-1 was a pipelined vector processor, not a multiple-processor computer, but it was capable of more than 100 million floating point operations per second.

Supercomputers have typically cost $10 million or more. The high cost of supercomputers once meant they were found almost exclusively in government research facilities, such as Los Alamos National Laboratory.

Over time, however, supercomputers began to appear outside of government facilities. In the late 1970s supercomputers showed up in capital-intensive industries. Petroleum companies harnessed supercomputers to help them look for oil, and automobile manufacturers started using these systems to improve the fuel efficiency and safety of their products.

Ten years later, hundreds of corporations around the globe were using supercomputers to support their business enterprises. The reason is simple: for many

businesses, quicker computations lead to a competitive advantage. More rapid crash simulations can reduce the time an automaker needs to design a new car. Faster drug design can increase the number of patents held by a pharmaceutical firm. High-speed computers have even been used to design products as mundane as disposable diapers!

Computing speeds have risen dramatically in the past 50 years. The ENIAC could perform about 350 multiplications per second. Today's supercomputers are more than a billion times faster, able to perform trillions of floating point operations per second.

Single processors are about a million times faster than they were 50 years ago. Most of the speed increase is due to higher clock rates that enable a single operation to be performed more quickly. The remaining speed increase is due to greater system concurrency: allowing the system to work simultaneously on multiple operations. The history of computing has been marked by rapid progress on both these fronts, as exemplified by contemporary high-performance microprocessors. Intel's Pentium 4 CPU, for example, has clock speeds well in excess of 1 GHz, two arithmetic-logic units (ALUs) clocked at twice the core processor clock speed, and extensive hardware support for out-of-order speculative execution of instructions.

How can today's supercomputers be a billion times faster than the ENIAC, if individual processors are only about a million times faster? The answer is simple: the remaining thousand-fold speed increase is achieved by collecting thousands of processors into an integrated system capable of solving individual problems faster than a single CPU; i.e., a parallel computer.

The meaning of the word *supercomputer*, then, has changed over time. In 1976 *supercomputer* meant a Cray-1, a single-CPU computer with a high-performance pipelined vector processor connected to a high-performance memory system. Today, *supercomputer* means a parallel computer with thousands of CPUs.

The invention of the microprocessor is a watershed event that led to the demise of traditional minicomputers and mainframes and spurred the development of low-cost parallel computers. Since the mid-1980s, microprocessor manufacturers have improved the performance of their top-end processors at an annual rate of 50 percent while keeping prices more or less constant [90]. The rapid increase in microprocessor speeds has completely changed the face of computing. Microprocessor-based servers now fill the role formerly played by minicomputers constructed out of gate arrays or off-the-shelf-logic. Even mainframe computers are being constructed out of microprocessors.

1.4 MODERN PARALLEL COMPUTERS

Parallel computers only became attractive to a wide range of customers with the advent of Very Large Scale Integration (VLSI) in the late 1970s. Supercomputers such as the Cray-1 were far too expensive for most organizations. Experimental parallel computers were less expensive than supercomputers, but they were still relatively costly. They were unreliable, to boot. VLSI technology allowed

computer architects to reduce the chip count to the point where it became possible to construct affordable, reliable parallel systems.

1.4.1 The Cosmic Cube

In 1981 a group at Caltech led by Charles Seitz and Geoffrey Fox began work on the Cosmic Cube, a parallel computer constructed out of 64 Intel 8086 microprocessors [34]. They chose the Intel 8086 because it was the only microprocessor available at the time that had a floating-point coprocessor, the Intel 8087. The complete 64-node system became operational in October 1983, and it dramatically illustrated the potential for microprocessor-based parallel computing. The Cosmic Cube executed its application programs at about 5 to 10 million floating point operations per second (5 to 10 **megaflops**). This made the Cosmic Cube 5 to 10 times the speed of a Digital Equipment Corporation VAX 11/780, the standard research minicomputer of the day, while the value of its parts was less than half the price of a VAX. In other words, the research group realized a price-performance jump of between 10 and 20 times by running their programs on a "home-made" parallel computer rather than a VAX. The Cosmic Cube was reliable, too; it experienced only two hard failures in its first year of operation.

Intel Corporation had donated much of the hardware for the Cosmic Cube. When it sent employee John Palmer to Caltech to see what Seitz and Fox had done, Palmer was so impressed he left Intel to start his own parallel computer company, nCUBE. Intel's second delegation, led by Justin Rattner, was equally impressed. Rattner became the technical leader of a new Intel parallel computer division called Intel Scientific Supercomputing.

1.4.2 Commercial Parallel Computers

Commercial parallel computers manufactured by Bolt, Beranek and Newman (BBN) and Denelcor were available before the Cosmic Cube was completed, but the Cosmic Cube stimulated a flurry of new activity. Table 1.1 is a list of just a few of the many organizations that jumped into the fray [116].

Companies from around the world began selling parallel computers. Intel's Supercomputer Systems Division and small start-up firms, such as Meiko, nCUBE, and Parsytec, led the way, while more established computer companies (IBM, NEC, and Sun Microsystems) waited until the field had become more mature. It is interesting to note that even Cray Research, Inc., famous for its custom, very high-performance, pipelined CPUs, eventually introduced a microprocessor-based parallel computer, the T3D, in 1993.

Other companies produced parallel computers with a single CPU and thousands of arithmetic-logic units (ALUs) implemented in VLSI. The most famous of these computers was the Connection Machine, built by Thinking Machines Corporation. This massively parallel computer, first shipped in 1986, contained 65,536 single-bit ALUs.

By the mid-1990s most of the companies on our list had either gotten out of the parallel computer business, gone bankrupt, or been purchased by larger firms. Despite the industry shakeout, leading computer manufacturers such as

Table 1.1 Some of the organizations that delivered commercial parallel computers based on microprocessor CPUs in the 10-year period 1984–1993 and their current status.

Company	Country	Year	Status in 2001
Sequent	U.S.	1984	Acquired by IBM
Intel	U.S.	1984	Out of the business*
Meiko	U.K.	1985	Bankrupt
nCUBE	U.S.	1985	Out of the business
Parsytec	Germany	1985	Out of the business
Alliant	U.S.	1985	Bankrupt
Encore	U.S.	1986	Out of the business
Floating Point Systems	U.S.	1986	Acquired by Sun
Myrias	Canada	1987	Out of the business
Ametek	U.S.	1987	Out of the business
Silicon Graphics	U.S.	1988	Active
C-DAC	India	1991	Active
Kendall Square Research	U.S.	1992	Bankrupt
IBM	U.S.	1993	Active
NEC	U.S.	1993	Active
Sun Microsystems	U.S.	1993	Active
Cray Research	U.S.	1993	Active (as Cray Inc.)

*"Out of the business" means the company is no longer selling general-purpose parallel computer systems.

Hewlett-Packard, IBM, Digital Equipment Corporation, Silicon Graphics, and Sun Microsystems all had parallel computers in their product lines by the mid-1990s.

These commercial systems ranged in price from several hundred thousand dollars to several million dollars. Compared to a commodity PC, the price per CPU in a commercial parallel computer was high, because these systems contained custom hardware to support either shared memory or low-latency, high-bandwidth interprocessor communications.

Some commercial parallel computers had support for higher-level parallel programming languages and debuggers, but the rapid evolution in the underlying hardware of parallel systems, even those manufactured by the same vendor, meant their systems programmers were perpetually playing catch-up. For this reason systems programming tools for commercial parallel computers were usually primitive, with the consequence that researchers found themselves programming these systems using the "least common denominator" approach of C or FORTRAN combined with a standard message-passing library, typically PVM or MPI. Vendors focused their efforts on penetrating the large commercial market, rather than serving the needs of the relatively puny scientific computing market. Hence computational scientists seeking peak performance from commercial parallel systems often felt they received inadequate support from vendors, and so they adopted a do-it-yourself attitude.

1.4.3 Beowulf

Meanwhile, driven by the popularity of personal computing for work and entertainment, PCs became a commodity market, characterized by rapidly improving

performance and razor-thin profit margins. The dynamic PC marketplace set the stage for the next breakthrough in parallel computing.

In the summer of 1994, at NASA's Goddard Space Flight Center, Thomas Sterling and Don Becker built a parallel computer entirely out of commodity hardware and freely available software. Their system, named Beowulf, contained 16 Intel DX4 processors connected by multiple 10 Mbit/sec Ethernet links. The cluster ran the Linux operating system, used GNU compilers, and supported parallel programming with the MPI message-passing library—all freely available software.

The high-performance computing research community rapidly embraced the Beowulf philosophy. At the Supercomputing '96 conference, both NASA and the Department of Energy demonstrated Beowulf clusters costing less than $50,000 that achieved greater than 1 billion floating point operations per second (1 **gigaflop**) performance on actual applications. At the Supercomputing '97 conference, Caltech demonstrated a 140-node cluster running an n-body simulation at greater than 10 gigaflops.

Beowulf is an example of a system constructed *out of* commodity, off-the-shelf (**COTS**) components. Unlike commercial systems, commodity clusters typically are not balanced between compute speed and communication speed: the communication network is usually quite slow compared to the speed of the processors. However, for many applications that are dominated by computations, clusters can achieve much better performance per dollar than commercial parallel computers. Because the latest CPUs typically appear in PCs months before they are available in commercial parallel computers, it is possible to construct a commodity cluster with newer, higher-performance CPUs than those available in a commercial parallel system. Commodity clusters have the additional, significant advantage of a low entry cost, which has made them a popular platform for academic institutions.

1.4.4 Advanced Strategic Computing Initiative

Meanwhile, the United States government has created an ambitious plan to build a series of five supercomputers costing up to $100 million each. This effort is motivated by the moratorium on underground nuclear testing signed by President Bush in 1992 and extended by President Clinton in 1993, as well as the decision by the United States to halt production of new nuclear weapons. As a result, the U.S. plans to maintain its stockpile of existing weapons well beyond their originally planned lifetimes. Sophisticated numerical simulations are required to guarantee the safety, reliability, and performance of the nuclear stockpile. The U.S. Department of Energy's Advanced Strategic Computing Initiative (**ASCI**) is developing a series of ever-faster supercomputers to execute these simulations.

The first of these supercomputers, ASCI Red, was delivered to Sandia National Laboratories in 1997. With just over 9,000 Intel Pentium II Xeon CPUs, it was the first supercomputer to sustain more than 1 trillion operations per second (1 **teraop**) on production codes. (Intel dropped out of the supercomputer business after delivering this system.) Lawrence Livermore National Laboratory in

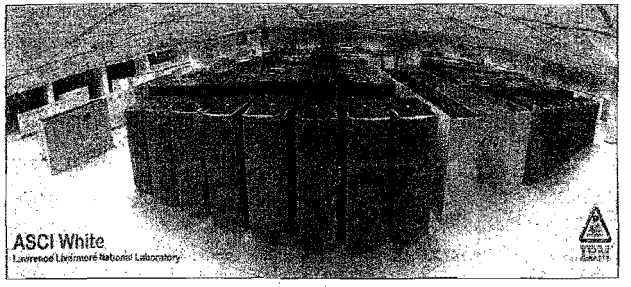


Figure 1.2 The ASCI White supercomputer at Lawrence Livermore National Laboratory contains 8192 PowerPC CPUs and is capable of sustaining more than 10 trillion operations per second on production programs. It was the fastest computer in the world in the year 2000. (Photo courtesy Lawrence Livermore National Laboratory)

California received delivery of the second supercomputer in the series, ASCI Blue Pacific, from IBM in 1998. It consists of 5,856 PowerPC CPUs and is capable of sustained performance in excess of 3 teraops.

In 2000 IBM delivered the third ASCI supercomputer, ASCI White, to the Lawrence Livermore National Laboratory (Figure 1.2). ASCI White actually is composed of three separate systems. The production system is an SMP-based multicomputer. It has 512 nodes; each node is an SMP with 16 PowerPC CPUs. The aggregate speed of the 8,192 CPUs has been benchmarked at more than 10 teraops on a computation of interest.

If the U.S. Department of Energy maintains this pace, tripling the performance of its ASCI supercomputers every two years, it will meet its goal of installing a 100 teraops computer by 2004.

1.5 SEEKING CONCURRENCY

As we have seen, parallel computers are more available than ever, but in order to take advantage of multiple processors, programmers and/or compilers must be able to identify operations that may be performed in parallel (i.e., concurrently).

1.5.1 Data Dependence Graphs

A formal way to identify parallelism in an activity is to draw a data dependence graph. A **data dependence graph** is a directed graph in which each vertex represents a task to be completed. An edge from vertex u to vertex v means that task u must be completed before task v begins. We say that "Task v is dependent on task u." If there is no path from u to v, then the tasks are **independent** and may be performed concurrently.

As an analogy, consider the problem of performing an estate's weekly landscape maintenance. Allan is leader of an eight-person crew working for Speedy

Landscape, Inc. (Figure 1.3a). His goal is to complete the four principal tasks—mowing the lawns, edging the lawns, weeding the gardens, and checking the sprinklers—as quickly as possible. Mowing must be completed before the sprinklers are checked. (Think of this as a dependence involving the four lawns as shared "variables." The lawns may take on the value "wet and cut" only after they have taken on the value "cut.") Similarly, edging and weeding must be completed before the sprinklers are checked. However, mowing, edging, and weeding may be done concurrently. Someone must also turn off the security system before the crew enters the estate and turn the system back on when the crew leaves. Allan represents these tasks using a dependence graph (Figure 1.3b).

Knowing the relative sizes of the respective jobs and the capabilities of his employees, Allan decides four crew members should mow the lawn while two crew members edge the lawn and two other crew members weed the gardens (Figure 1.3c).

Three different task patterns appear in Figure 1.3c. Figure 1.4 illustrates each of these patterns in isolation. The labels inside the circles represent the kinds of tasks being performed. Multiple circles with the same label represent tasks performing the same operation on different operands.

1.5.2 Data Parallelism

A data dependence graph exhibits **data parallelism** when there are independent tasks applying the same operation to different elements of a data set (Figure 1.4a).

Here is an example of fine-grained data parallelism embedded in a sequential algorithm:

```
for i ← 0 to 99 do
  a[i] ← b[i] + c[i]
endfor
```

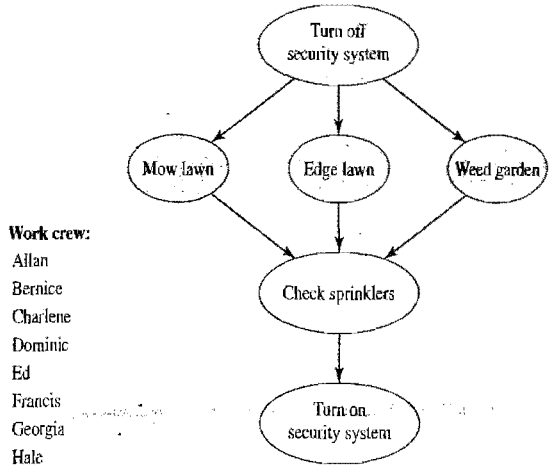
The same operation—addition—is being performed on the first 100 elements of arrays b and c, with the results being put in the first 100 elements of a. All 100 iterations of the loop could be executed simultaneously.

1.5.3 Functional Parallelism

A data dependence graph exhibits **functional parallelism** when there are independent tasks applying different operations to different data elements (Figure 1.4b).

Here is an example of fine-grained functional parallelism embedded in a sequential algorithm:

```
a ← 2
b ← 3
m ← (a + b)/2
s ← (a² + b²)/2
v ← s − m²
```



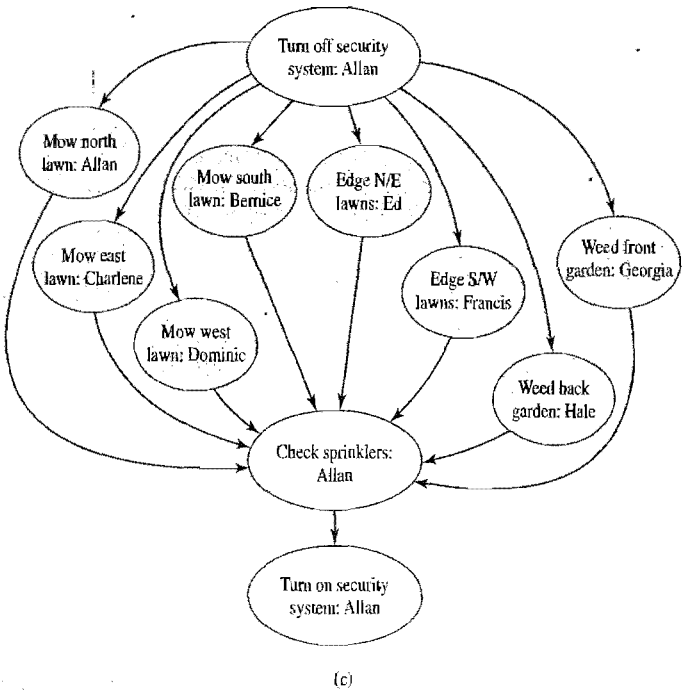


Figure 1.3 Most realistic problems have data parallelism, functional parallelism, and precedence constraints between tasks. (a) An eight-person work crew is responsible for landscape maintenance at Medici Manor. (b) A data dependence graph shows which tasks must be completed before others begin. If there is no path from vertex u to vertex v, the tasks may proceed concurrently (functional parallelism). (c) The larger tasks have been divided into subtasks, in which several employees perform the same activity on different portions of the estate (data parallelism).

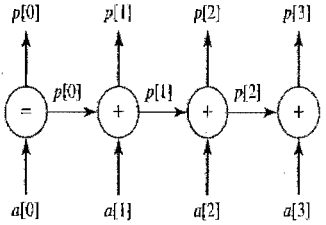


Figure 1.6 A pipeline to compute partial sums. Each circle represents a process. The leftmost stage inputs $a[0]$, outputs its value as $p[0]$, and passes $p[0]$ to the next stage. All other stages i input $a[i]$, collect $p[i-1]$ from their predecessors, add the two values, and output the sum as $p[i]$.

this book, we'll be looking for sources of parallelism in problems requiring fa more computations.

1.6 DATA CLUSTERING

Let's consider a practical example of a computationally intensive problem an try to find opportunities for parallelism.

Modern computer systems are capable of collecting and storing extraordinar amounts of data. For example, the World Wide Web contains hundreds of millior of pages. U.S. Census data constitute another very large dataset. Using a comput system to access salient facts or detect meaningful patterns is variously calle **data mining** or **scientific data analysis**. Data mining is a compute-intensiv "off-line" operation, in contrast to data retrieval, which is an I/O-intensive, "or line" operation.

Multidimensional data clustering is an important tool for data mining. **Dat clustering** is the process of organizing a dataset into groups, or clusters, c "similar" items. Clustering makes it easier to find additional items closely re lated to an item of interest.

Suppose we have a collection of N text documents. We will examine eac document to come up with an estimate of how well it covers each of D differer topics, and we will assign each document to one of K different clusters, where eac cluster contains "similar" documents. See Figure 1.7. A performance functio indicates how well clustered the documents are. Our goal is to optimize the valu of the performance function.

Figure 1.8 contains a high-level description of a sequential algorithm to solv the data clustering problem. How could parallelism be used to speed the executio of this algorithm?

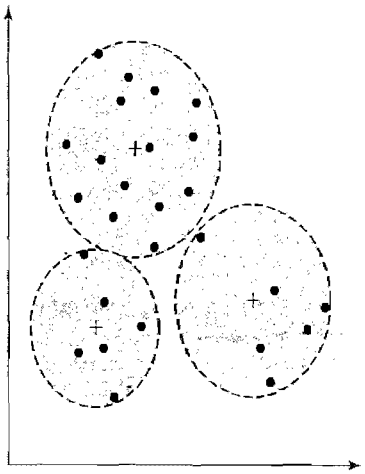


Figure 1.7 An example of document clustering for which $N = 20$, $D = 2$, and $K = 3$. There are 20 documents (represented by black dots). We measure each document's coverage of two topics (hence each document is represented by a point in two-dimensional space). The documents are organized into three clusters (centered around the crosses). Can you find a better clustering?

Data Clustering:

1. Input N documents
2. For each of the N documents generate a D-dimensional vector indicating how well it covers the D different topics
3. Choose the K initial cluster centers using a random sample
4. Repeat the following steps for I iterations or until the performance function converges, whichever comes first:
   (a) For each of the N documents, find the closest center and compute its contribution to the performance function
   (b) Adjust the K cluster centers to try to improve the value of the performance function
5. Output K centers

Figure 1.8 A sequential algorithm to find K centers that optimally categorize N documents.

Our first step in the analysis is to draw a data dependence graph. While we could draw one vertex for each step in the pseudocode algorithm, it is better to draw a vertex for each step of the algorithm *for each document or cluster center*, because it exposes more opportunities for parallelism. The resulting data dependence graph appears in Figure 1.9.

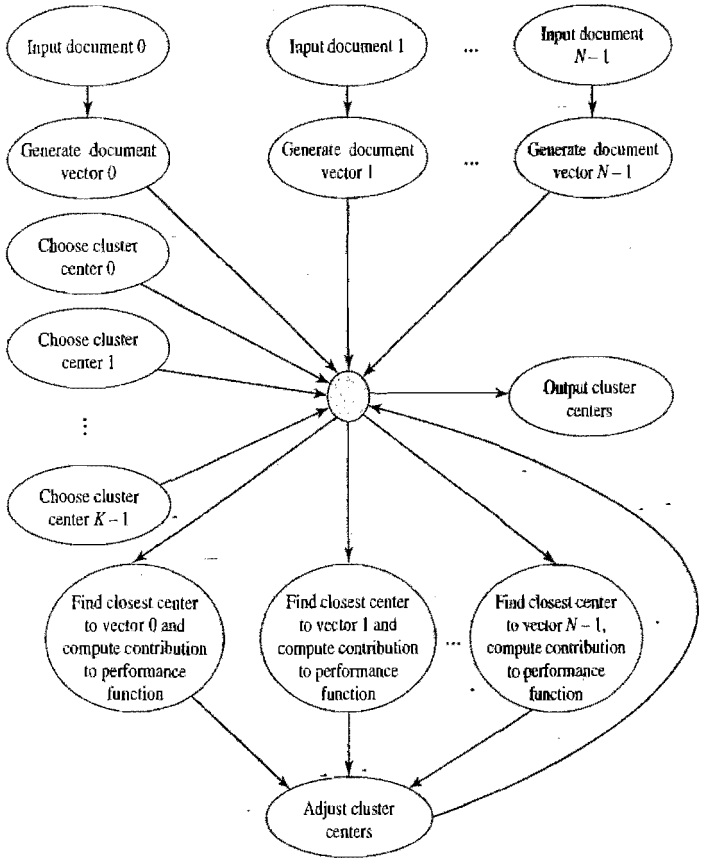


Figure 1.9 Dependence diagram for the document clustering algorithm. The small, unlabeled vertex in the middle is a "null task" containing no operations. Its purpose is to reduce the number of directed edges and make the diagram easier to read.

A good data dependence graph makes data and functional parallelism easy to find. That's certainly true in this case.

First, let's list the opportunities for data parallelism:

- Each document may be input in parallel.
- Each document vector may be generated in parallel.
- The original cluster centers may be generated in parallel.
- The closest cluster center to each document vector and that vector's contribution to the overall performance function may be computed in parallel.

Next, let's look for functional parallelism. The only independent sets of vertices are those representing the document input and vector generation tasks and those representing the center generation tasks. These two sets of tasks could be performed concurrently.

After we have identified parallelism in a problem, our next step is to develop an algorithm implemented in a programming language. Let's take a look at the variety of ways in which parallel computers have been programmed.

1.7 PROGRAMMING PARALLEL COMPUTERS

In 1988 McGraw and Axelrod identified four distinct paths for the development of applications software for parallel computers [85]:

1. Extend an existing compiler to translate sequential programs into parallel programs.
2. Extend an existing language with new operations that allow users to express parallelism.
3. Add a new parallel language layer on top of an existing sequential language.
4. Define a totally new parallel language and compiler system.

Let's examine the advantages and disadvantages of each of these alternatives.

1.7.1 Extend a Compiler

One approach to the problem of programming parallel computers is to develop parallelizing compilers that can detect and exploit the parallelism in existing programs written in a sequential language.

Much research has been done into the parallel execution of functional or logic programs, which can contain a good deal of intrinsic parallelism. However, most of the focus has been on the imperative programming language Fortran. Proponents of the development of parallelizing compilers for Fortran point out that existing Fortran programs represent the investment of billions of dollars and millenia of programmer effort. While not all of these programs would benefit from execution on a parallel computer, some organizations (such as the national laboratories run by the U.S. Department of Energy) would like to speed the execution of many sophisticated Fortran codes. The time and labor that could be saved from the automatic parallelization of these programs makes this approach highly desirable. In addition, parallel programming is more difficult than programming in Fortran, leading to higher program development costs. For these reasons some believe it makes more sense for programmers to continue to use simpler, sequential languages, leaving the parallelization up to a compiler.

The development of parallelizing compilers has been an active area of research for more than two decades, and many experimental systems have been developed. Companies such as Parallel Software Products have begun offering

compilers that translate Fortran 77 code into parallel programs targeted for either message-passing or shared-memory architectures.

This approach does have its detractors. For example, Hatcher and Quinn point out that the use of a sequential imperative language "pits the programmer against the compiler in a game of hide and seek. The algorithm may have a certain amount of inherent parallelism. The programmer hides the parallelism in a sea of DO loops and other control structures, and then the compiler must seek it out. Because the programmer may have to specify unneeded sequentializations when writing programs in a conventional imperative language, some parallelism may be irretrievably lost" [49].

One response to these concerns is to allow the programmer to annotate the sequential program with compiler directives. These directives provide information to the compiler that may help it correctly parallelize program segments.

1.7.2 Extend a Sequential Programming Language

A much more conservative approach to developing a parallel programming environment is to extend a sequential programming language with functions that allow the programmer to create and terminate parallel processes, synchronize them, and enable them to communicate with each other. There must also be a way to distinguish between public data (shared among the processes) and private data (for which each process has a copy).

Extending a sequential programming language is the easiest, quickest, least expensive, and (perhaps for these reasons) the most popular approach to parallel programming, because it simply requires the development of a subroutine library. The existing language and hence its compiler can be used as is. The relative ease with which libraries can be developed enables them to be constructed rapidly for new parallel computers. For example, libraries meeting the MPI standard exist for virtually every kind of parallel computer. Hence programs written with MPI function calls are highly portable.

Giving programmers access to low-level functions for manipulating parallel processors provides them with maximum flexibility with respect to program development. Programmers can implement a wide variety of parallel designs using the same programming environment.

However, because the compiler is not involved in the generation of parallel code, it cannot flag errors. The lack of compiler support means that the programmer has no assistance in the development of parallel codes. It is surprisingly easy to write parallel programs that are difficult to debug.

Consider these comments from parallel programming pioneers circa 1988:

"Suddenly, even very simple tasks, programmed by experienced programmers who were dedicated to the idea of making parallel programming a practical reality, seemed to lead inevitably to upsetting, unpredictable, and totally mystifying bugs" (Robert B. Babb II) [6].

"The behavior of even quite short parallel programs can be astonishingly complex. The fact that a program functions correctly once, or even one hundred times,

with some particular set of inputs, is no guarantee that it will not fail tomorrow with the same inputs" (James R. McGraw and Timothy S. Axelrod) [85].

1.7.3 Add a Parallel Programming Layer

You can think of a parallel program as having two layers. The lower layer contains the core of the computation, in which a process manipulates its portion of the data to produce its portion of the result. An existing sequential programming language would be suitable for expressing this portion of the activity. The upper layer controls the creation and synchronization of processes and the partitioning of the data among the processes. These actions could be programmed using a parallel language (perhaps a visual programming language). A compiler would be responsible for translating this two-layer parallel program into code suitable for execution on a parallel computer.

Two examples of this approach are the Computationally Oriented Display Environment (CODE) and the Heterogeneous Network Computing Environment (Hence). These systems allow the user to depict a parallel program as a directed graph, where nodes represent sequential procedures and arcs represent data dependences among procedures [12].

This approach requires the programmer to learn and use a new parallel programming system, which may be the reason it has not captured much attention in the parallel programmer community. While research prototypes are being distributed, the author knows of no commercial systems based on this philosophy.

1.7.4 Create a Parallel Language

The fourth approach is to give the programmer the ability to express parallel operations explicitly.

One way to support explicit parallel programming is to develop a parallel language from scratch. The programming language occam is a famous example of this approach. With a syntax strikingly different from traditional imperative languages, it supports parallel as well as sequential execution of processes and automatic process communication and synchronization.

Another way to support explicit parallelism is to add parallel constructs to an existing language. Fortran 90, High Performance Fortran, and C* are examples of this approach.

Fortran 90 is an ANSI and ISO standard programming language, the successor to Fortran 66 and Fortran 77. (Outside of the United States, Fortran 90 replaced Fortran 77. Within the U.S., Fortran 90 is viewed as an additional standard.) It contains many features not incorporated in Fortran 77, including array operations. Fortran 90 allows entire, multidimensional arrays to be manipulated in expressions. For example, suppose A, B, and C are arrays of real variables having 10 rows and 20 columns, and we want to add A and B, assigning the sum to C. In Fortran 77 we would need to write a doubly nested DO loop to accomplish

1.14 Suppose we are going to speed the execution of the data clustering algorithm by using p processors to generate the D-dimensional vectors for each of the N documents. One approach would be to preallocate about N/p documents to each processor. Another approach would be to put the documents on a list and let processors remove documents as fast as they could process them. Discuss one advantage of each approach.

1.15 Consider the vector-generation step of the data clustering algorithm described in this chapter. Assume the time required to perform this step is directly proportional to document size. Suggest an approach for allocating documents to processors that may avoid the problems associated with either preallocating N/p documents to each processor or having processors retrieve unprocessed documents from a central list.

CHAPTER

2

Parallel Architectures

What king marching to war against another king would not first sit down and consider whether with ten thousand men he could stand up to the other who advanced against him with twenty thousand?

Luke 14:31

2.1 INTRODUCTION

In the roughly three decades between the early 1960s and the mid-1990s, scientists and engineers explored a wide variety of parallel computer architectures. Development reached a zenith in the 1980s. Some companies took advantage of newly available VLSI fabrication facilities to develop custom processors for parallel computers, while others relied upon the same general-purpose CPUs used in workstations and personal computers. Experts passionately debated whether the dominant parallel computer systems would contain at most a few dozen high-performance processors or thousands of less-powerful processors.

Today, many of the hotly debated questions have been resolved. Systems containing thousands of primitive processors are a rarity. The performance of custom-designed processors could not keep up with the rapid gains made by commodity processors. As a result, most contemporary parallel computers are constructed out of commodity CPUs.

This chapter is a brief overview of parallel computer architectures. We begin by examining a variety of interconnection networks that can be used to link processors in a parallel system. We present processor arrays, multiprocessors, and multicomputers, the three most popular parallel computer architectures in the past two decades. We discuss ways to organize a commodity cluster (a particular kind of multicomputer), and we explain what makes a commodity cluster different from a network of workstations. We introduce Flynn's famous

taxonomy of serial and parallel computers and take a brief look at systolic arrays, a heavily pipelined architecture that has not been widely adopted.

2.2 INTERCONNECTION NETWORKS

All computers with multiple processors must provide a way for processors to interact. In some systems processors use the interconnection network to access a shared memory. In other systems processors use the interconnection network to send messages to each other. This section outlines the two principal types of interconnection media and presents several popular topologies for switch networks.

2.2.1 Shared versus Switched Media

Processors in a parallel computer may communicate over shared or switched interconnection media. A **shared medium** allows only one message at a time to be sent (Figure 2.1a). Processors broadcast their messages over the medium. Each processor "listens" to every message and receives the ones for which it is the destination. Ethernet is a well-known example of a shared medium.

Typically, arbitration for access to a shared medium is decentralized among the processors. Before sending a message, a processor "listens" until the medium is unused, then attempts to send its message. If two processors attempt to send messages simultaneously, the messages are garbled and must be resent. The processors wait a random amount of time and then attempt once again to send their messages. Message collisions can significantly degrade the performance of a heavily utilized shared medium.

In contrast, **switched interconnection media** support point-to-point messages among pairs of processors (Figure 2.1b). Each processor has its own communication path to the switch. Switches have two important advantages over

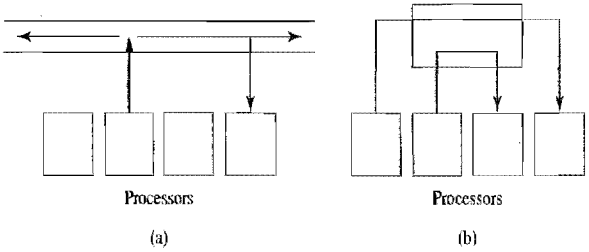


Figure 2.1 Contrasting shared versus switched media. (a) A shared medium allows only one message at a time to be sent. Each processor "listens" to every message and receives the ones for which it is the destination. (b) A switched medium supports the simultaneous transmission of multiple messages among different pairs of processors.

shared media. They support the concurrent transmission of multiple messages among different pairs of processors, and they support the scaling of the interconnection network to accommodate greater numbers of processors.

2.2.2 Switch Network Topologies

A switch network can be represented by a graph in which nodes represent processors and switches and edges represent communication paths. Each processor is connected to one switch. Switches connect processors and/or other switches.

In a **direct topology** the ratio of switch nodes to processor nodes is 1:1. Every switch node is connected to one processor node and one or more other switch nodes. In an **indirect topology** the ratio of switch nodes to processor nodes is greater than 1:1. Some of the switches simply connect other switches.

We can evaluate switch network topologies according to criteria that help us understand their effectiveness in implementing efficient parallel algorithms on real hardware. These criteria are:

- Diameter: The **diameter** of a network is the largest distance between two switch nodes. Low diameter is better, because the diameter puts a lower bound on the complexity of parallel algorithms requiring communication between arbitrary pairs of nodes.
- Bisection width: The **bisection width** of a switch network is the minimum number of edges between switch nodes that must be removed in order to divide the network into two halves (within one). High bisection width is better, because in algorithms requiring large amounts of data movement, the size of the data set divided by the bisection width puts a lower bound on the complexity of the parallel algorithm. Proving the bisection width of a network is often more difficult than a cursory visual inspection might lead you to believe.
- Edges per switch node: It is best if the number of **edges per switch node** is a constant independent of the network size, because then the processor organization scales more easily to systems with large numbers of nodes.
- Constant **edge length:** For scalability reasons it is best if the nodes and edges of the network can be laid out in three-dimensional space so that the maximum edge length is a constant independent of the network size.

Many switch network topologies have been analyzed. We focus on six of them: 2-D mesh, binary tree, hypertree, butterfly, hypercube, and shuffle-exchange. The 2-D mesh, hypertree, butterfly, and hypercube have appeared in commercial parallel computers; the binary tree and shuffle-exchange networks are presented as interesting points in the design space.

2.2.3 2-D Mesh Network

The two-dimensional mesh network (shown in Figure 2.2) is a direct topology in which the switches are arranged into a two-dimensional lattice. Communication

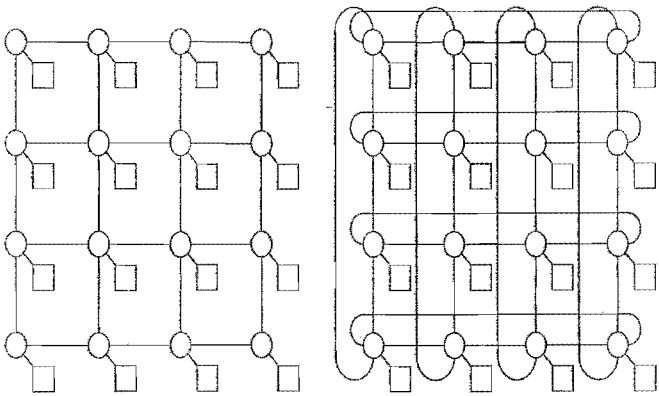


Figure 2.2 Variants of the 2-D mesh network. Circles represent switches, while squares represent processors. (a) Without wraparound connections. (b) With wraparound connections.

is allowed only between neighboring switches; hence interior switches commun cate with four other switches. Some variants of the mesh model allow wraparour connections between switches on the edge of the mesh.

Let's evaluate the 2-D mesh network according to our four criteria. We assur that the mesh has n switch nodes and no wraparound connections. The mesh ha minimum diameter and maximum bisection width when it is as square as possibl in which case its diameter and bisection width are both $\Theta(\sqrt{n})$. It has a consta number of edges per switch, and it is possible to build an arbitrarily large mes with constant edge length.

2.2.4 Binary Tree Network

In a **binary tree network**, communications among the $n = 2^d$ processor nod are supported by a binary tree of $2n - 1$ switches (Figure 2.3). Each process node is connected to a leaf of the binary tree. Hence the binary tree network an example of an indirect topology. The interior switch nodes have at most thr links: two to children and one to a parent node.

The binary tree switch network has low diameter: $2 \log n$.[1] However its b section width is the minimum value possible, 1. Assuming nodes occupy physic space, it is impossible to arrange the switch nodes of a binary tree network three-dimensional space such that as the number of nodes increases, the leng of the longest edge is always less than a specified constant.

[1] In this book $\log n$ means $\log_2 n$.

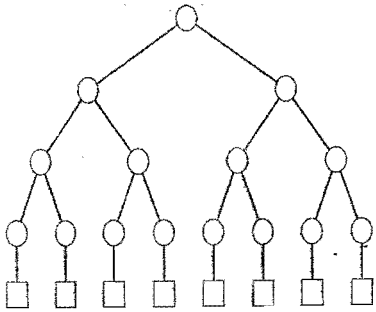


Figure 2.3 Binary tree network with eight processor nodes and 15 switch nodes.

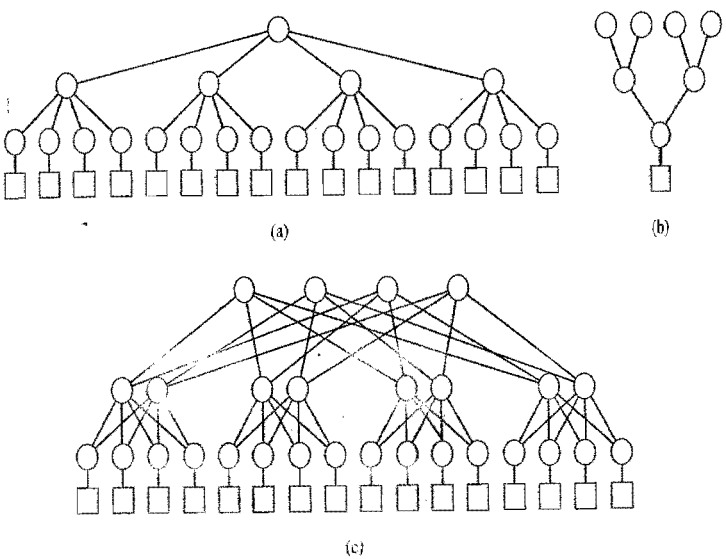


Figure 2.4 Hypertree network of degree 4 and depth 2. Circles represent switches, and squares represent processors. (a) Front view. (b) Side view. (c) Complete network.

2.2.5 Hypertree Network

A hypertree is an indirect topology that shares the low diameter of a binary tree but has an improved bisection width. The easiest way to think of a **hypertree network** of degree k and depth d is to consider the network from two different angles (Figure 2.4). From the front it looks like a complete k-ary tree of height d (Figure 2.4a). From the side, the same network looks like an upside-down binary tree of height d (Figure 2.4b). Joining the front and side views yields the complete network. Figure 2.4c illustrates a hypertree network of degree 4 and height 2.

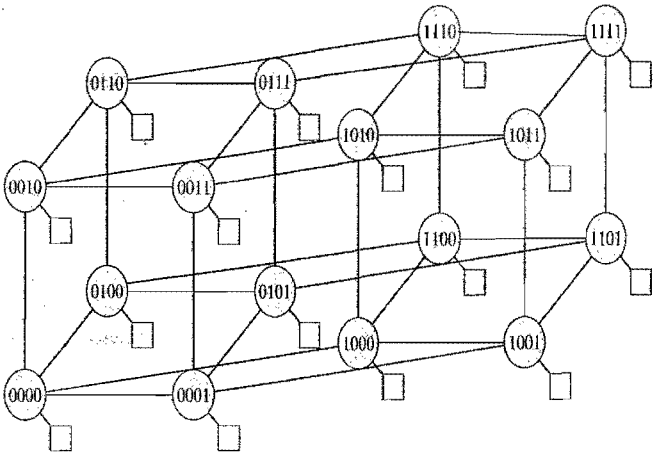


Figure 2.7 A hypercube network with 16 processor nodes and an equal number of switch nodes. Circles represent switches, and squares represent processors. Processor/switch pairs share addresses.

nodes are labeled $0, 1, \ldots, 2^d - 1$; two switch nodes are adjacent if their binary labels differ in exactly one bit position. A four-dimensional hypercube is shown in Figure 2.7.

The diameter of a hypercube with $n = 2^d$ switch nodes is $\log n$ and its bisection width is $n/2$. The hypercube organization has low diameter and high bisection width at the expense of the other two factors we are considering. The number of edges per switch node is $\log n$ (we're not counting the edge to the processor), and the length of the longest edge in a hypercube network increases as the number of nodes in the network increases.

Let's explore how to route messages in a hypercube. Take another look at Figure 2.7, and note how edges always connect switches whose addresses differ in exactly one bit position. For example, links connect switch 0101 with switches 1101, 0001, 0111, and 0100. Knowing this, we can easily find a shortest path between the source and destination switches.

Suppose we want to send a message from switch 0101 to switch 0011 in a four-dimensional hypercube. The addresses differ in two bit positions. That means the shortest path between the two switches has length 2. Here is a shortest path:

$$0101 \rightarrow 0001 \rightarrow 0011$$

Can you think of another path from 0101 to 0011 that also has length 2? Changing the order in which we flip the bits that differ results in a different path:

$$0101 \rightarrow 0111 \rightarrow 0011$$

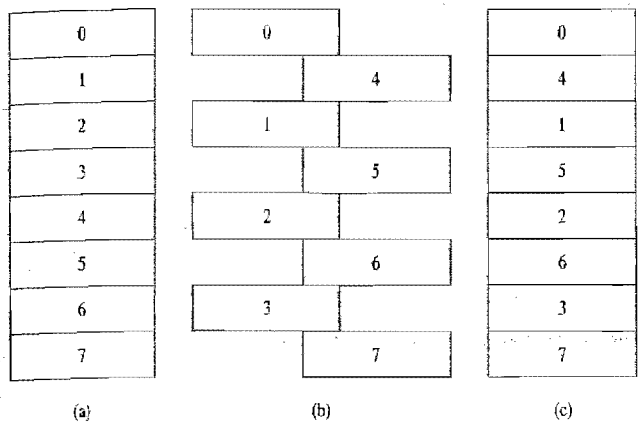


Figure 2.8 Derivation of the perfect shuffle permutation. (a) A sorted deck of cards. (b) The cards are shuffled perfectly. (c) The result of the perfect shuffle.

2.2.8 Shuffle-Exchange Network

The final network topology we are considering is based on the idea of a *perfect shuffle*. Imagine taking a sorted deck of cards (Figure 2.8a), dividing it exactly in half, and shuffling the two halves perfectly (Figure 2.8b). The resulting permutation of the original card ordering is called a **perfect shuffle** (Figure 2.8c). If we represent the original position of each card as a binary number, its new position can be calculated by performing a left cyclic rotation of the binary number. In other words, we shift every bit left one position, but the leftmost bit wraps around to the rightmost position. For example, card 5, originally at index 5 (101), ends up at index 3 (011).

A **shuffle-exchange network** is a direct topology with $n = 2^d$ processor/switch pairs. The pairs are numbered $0, 1, \ldots, n-1$. The switches have two kinds of connections, called shuffle and exchange. **Exchange connections** link pairs of switches whose numbers differ in their least significant bit. Each **shuffle connection** links switch i with switch j, where j is the result of cycling the bits of i left one position. For example, in an eight-node network, a shuffle connection links switch 5 (101) with switch 3 (011). Figure 2.9 illustrates a shuffle-exchange network with 16 nodes.

Every switch in a shuffle-exchange network has a constant number of edges: two outgoing and two incoming (not counting the link to the processor). The length of the longest edge increases with the network size. The diameter of an n-switch shuffle-exchange network is $2 \log n - 1$. The bisection width is $\approx n / \log n$.

Let's think about routing messages through a shuffle-exchange network. A path of maximum length follows $\log n$ exchange links and $\log n - 1$ shuffle links. The worst-case scenario is routing a message from switch 0 to switch $n-1$ (or vice versa). Suppose $n = 16$. Routing a message from 0000 to 1111 requires

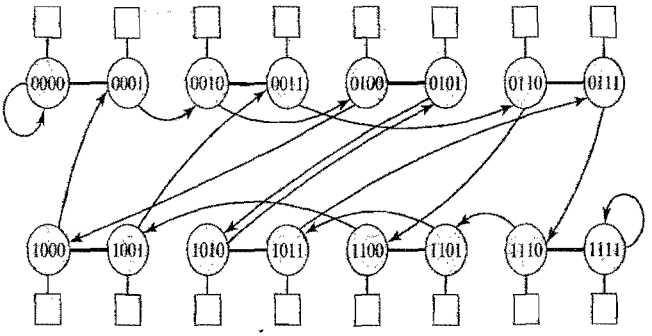


Figure 2.9 A shuffle-exchange network with 16 processor nodes and an equal number of switch nodes. Circles represent switches, and squares represent processors. Processor/switch pairs share addresses. The heavy black lines are the bidirectional exchange links, while the arrows represent unidirectional shuffle links.

$2 \log n - 1 = 7$ steps (where E and S refer to following exchange and shuffle links, respectively):

$$0000 \xrightarrow{E} 0001 \xrightarrow{S} 0010 \xrightarrow{E} 0011 \xrightarrow{S} 0110 \xrightarrow{E} 0111 \xrightarrow{S} 1110 \xrightarrow{E} 1111$$

It's not hard to devise an algorithm that always generates paths following $\log n - 1$ shuffle links, but skips exchange links in those cases where the low-order bit does not need to be flipped. For example, routing a message from 0011 to 0101 could be done by following three shuffle links and two exchange links:

$$0011 \xrightarrow{E} 0010 \xrightarrow{S} 0100 \xrightarrow{E} 0101 \xrightarrow{S} 1010 \xrightarrow{S} 0101$$

A more sophisticated algorithm would find even shorter paths by looking for patterns between the source and destination addresses, reducing the number of shuffle steps. For example, it would be able to recognize that the final two shuffles in the previous routing are unnecessary.

2.2.9 Summary

Table 2.1 summarizes the characteristics of the six interconnection networks described in this section. No network can be optimal in every regard. The 2-D mesh is the only network that maintains a constant edge length as the number of nodes increases, but it is also the only network that does not have logarithmic diameter. The butterfly and hypercube networks have high bisection width, but the butterfly network has $\Theta(n \log n)$ switch nodes, and the hypercube network is the only network in which the number of edges per node is not a constant. The shuffle-exchange network represents a design midpoint, with a fixed number of edges per node, low diameter, and good bisection width. The 4-ary hypertree is

Table 2.1 Attributes of six switching network topologies. The column labeled "Edges/nodes" refers to the maximum number of switches to which a switching node is connected.

	Processor nodes	Switch nodes	Diameter	Bisection width	Edges/ nodes	Constant edge length
2-D mesh	$n = d^2$	n	$2(\sqrt{n} - 1)$	$\sqrt{n}$	4	Yes
Binary tree	$n = 2^d$	$2n - 1$	$2 \log n$	1	3	No
4-ary hypertree	$n = 4^d$	$2n - \sqrt{n}$	$\log n$	$n/2$	6	No
Butterfly	$n = 2^d$	$n(\log n + 1)$	$\log n$	$n/2$	4	No
Hypercube	$n = 2^d$	n	$\log n$	$n/2$	$\log n$	No
Shuffle-exchange	$n = 2^k$	n	$2 \log n - 1$	$\approx n/\log n$	2	No

superior to the binary tree in nearly every respect, with fewer switch nodes, lower diameter, and high bisection width.

2.3 PROCESSOR ARRAYS

A **vector computer** is a computer whose instruction set includes operations on vectors as well as scalars. Generally there are two ways of implementing a vector computer. A **pipelined vector processor** streams vectors from memory to the CPU, where pipelined arithmetic units manipulate them. The Cray-1 and Cyber-205, early supercomputers, are well-known examples of pipelined vector processors. We do not consider these architectures further.

A **processor array** is a vector computer implemented as a sequential computer connected to a set of identical, synchronized processing elements capable of simultaneously performing the same operation on different data. Many pioneering parallel computer development efforts resulted in the construction of processor arrays. One motivation for this design was the relatively high price of a control unit [52]. Another key motivation for the construction of processor arrays was the observation that a large fraction of scientific computations are data parallel [50]. That, of course, is exactly how a processor array achieves its parallelism.

2.3.1 Architecture and Data-parallel Operations

Let's look at the architecture of a generic processor array. It is a collection of simple processing elements controlled by a front-end computer (Figure 2.10).

The front-end computer is a standard uniprocessor. Its primary memory contains the instructions being executed as well as data that are manipulated sequentially by the front end. The processor array is divided into many individual processor-memory pairs. Data that are manipulated in parallel are distributed among these memories. In order to perform a parallel operation, the front-end computer transmits the appropriate instruction to the processors in the processor array, which simultaneously execute the instruction on operands stored in their local memories. A control path (indicated by a dashed line in Figure 2.10) allows the front-end computer to broadcast instructions to the back-end processors.

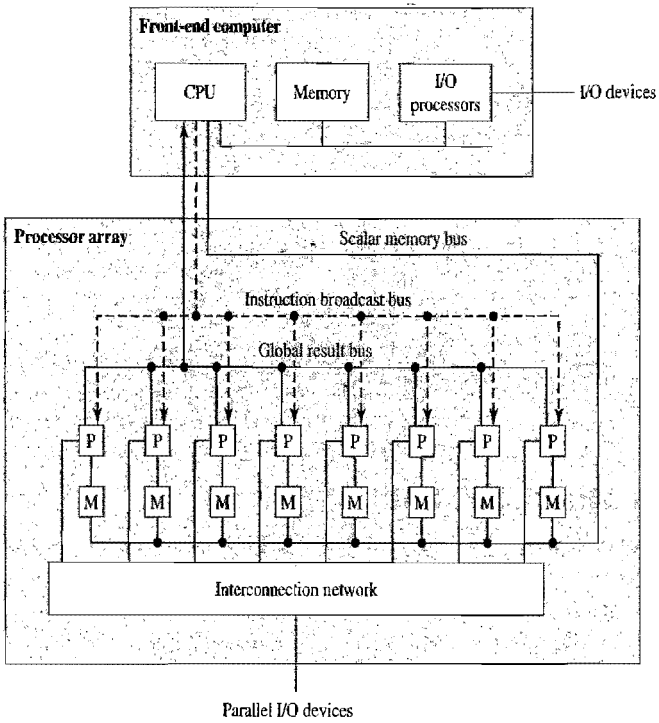


Figure 2.10 Architecture of a generic processor array. The processor array contains many primitive processors (shown by boxes labeled with a P). Each processor has its own memory (shown by boxes labeled with an M).

For example, suppose the processor array contains 1024 processors, labeled $p_0, p_1, \ldots, p_{1023}$. Imagine two 1024-element vectors A and B are distributed among the processors such that a_i and b_i are in the memory of processor p_i, for all i in the range $0 \leq i \leq 1023$. The processor array can perform the vector addition $A + B$ in a single instruction, because each processor p_i fetches its own pair of values a_i and b_i and performs the addition in parallel with all the other processors.

Note that the time spent by this processor array to add two 100-element vectors, or any two vectors of length ≤ 1024, is the same as the time needed to add two 1024-element vectors. The time needed to perform any particular instruction on the processor array is independent of the number of processors actually active.

What if the programmer wishes to manipulate a vector with more than 1024 elements? When the size of the vector exceeds the number of processors in the processor array, some or all of the processors need to store and manipulate multiple vector elements. For example, a 10,000-element vector can be stored on

a 1024-processor system by giving 784 processors 10 elements and 240 processors 9 elements: $784 \times 10 + 240 \times 9 = 10{,}000$. Depending upon the particular architecture and operating system, this mapping of vector elements to processors may or may not have to be managed by the programmer. For example, the operating system of Thinking Machines' Connection Machine supported the notion of virtual processors. The programmer could write programs manipulating vectors and matrices much larger than the size of the processor array. Microcoded instructions sent from the front end to the processor array managed the complexity of mapping virtual processors to physical processors.

2.3.2 Processor Array Performance

Performance is a metric indicating the amount of work accomplished per time unit. We can measure processor array performance in terms of operations per second. The performance of a processor array depends on the utilization of its processors. The size of the data structure being manipulated directly affects performance. Processor array performance is highest when all processors are active and the size of the data structure is a multiple of the number of processors.

EXAMPLE 2.1

Suppose a processor array contains 1024 processors. Each processor is capable of adding a pair of integers in 1 μsecond. What is the performance of this processor array adding two integer vectors of length 1024, assuming each vector is allocated to the processors in a balanced fashion?

■ Solution

The number of integer operations performed is 1024. Each processor performs one integer addition, requiring 1 μsecond.

$$\text{Performance} = \frac{1024 \text{ operations}}{1 \ \mu\text{second}} = 1.024 \times 10^9 \text{ operations/second}$$

EXAMPLE 2.2

Suppose a processor array contains 512 processors. Each processor is capable of adding a pair of integers in 1 μsecond. What is the performance of this processor array adding two integer vectors of length 600, assuming each vector is allocated to the processors in a balanced fashion?

■ Solution

The number of integer operations performed is 600. Since $600 > 512$, 88 processors must add two pairs of integers. The other 424 processors add only a single pair of integers. They sit idle while the other 88 processors add their second integer pair.

$$\text{Performance} = \frac{600 \text{ operations}}{2 \ \mu\text{second}} = 3 \times 10^8 \text{ operations/second}$$

2.3.3 Processor Interconnection Network

Of course, the typical parallel computation is far more complicated than simply adding two vectors. Often the new value of a vector or matrix element is a function of other elements, as in the implementation of a finite difference method to solve a partial differential equation:

$$a_i \leftarrow (a_{i-1} + a_{i+1})/2$$

To bring together operands stored in the memories of different processors, the processors can pass data through an interconnection network. The most popular interconnection network for processor arrays is the two-dimensional mesh. Besides the advantages previously noted, the two-dimensional mesh has the advantage of a relatively straightforward implementation in VLSI, where a single chip may contain a large number of processors (see Figure 2.11).

The interconnection network supports concurrent message passing. For example, in the two-dimensional mesh shown in Figure 2.11, each processing element can simultaneously send a value to the processing element to its "north."

2.3.4 Enabling and Disabling Processors

The processor array exhibits synchronous execution—that is, all the individual processors work in lockstep. However, it is possible for only a subset of the

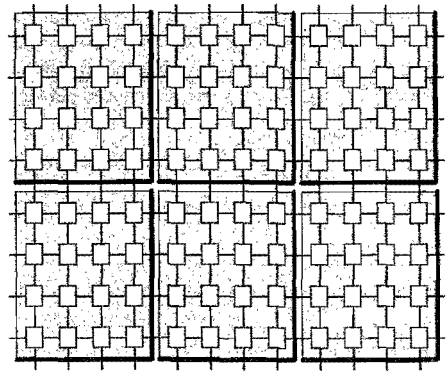


Figure 2.11 Illustration of how an 8 x 12 processor array with a two-dimensional mesh interconnection could be arranged to minimize wire lengths. A single VLSI chip contains 16 processing elements arranged in a 4 x 4 mesh. A 2 x 3 arrangement of chips produces the desired 96-processor array. This figure shows the interconnection network; it does not illustrate the connections between the processors and the front-end computer.

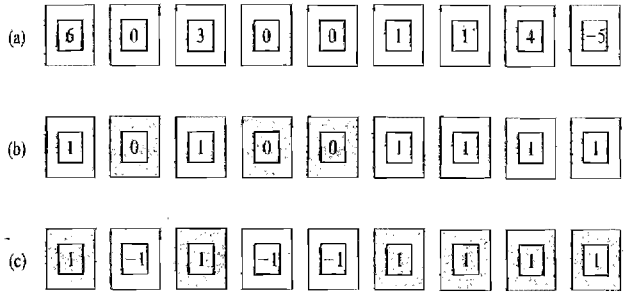


Figure 2.12 Execution of an **if-then-else** statement. (a) Vector A, with nine elements, is distributed among the memories of nine processors, one element per processor. If the value is nonzero, then it will be changed to 1, else it will be changed to -1. (b) Shading indicates the processors that are masked out (inactive) because their element of A is 0. The remaining processors set their values of A to 1. (c) The active and inactive processors switch roles. The active processors set their values of A to -1.

processors to perform an instruction. Each processor has a masking bit that allows it to "opt out" of performing an instruction. Masking is useful if the number of data items being manipulated is not an exact multiple of the size of the processor array.

Masking also enables the processor array to support conditionally executed parallel operations. For example, suppose integer vector A is distributed across the processor array, one element per processor, and we want to convert every nonzero value inside A to 1 and every 0 to -1 (Figure 2.12). First, every processor tests to see if its element of A has the value 0. If so, the processor sets its mask bit, indicating it is not executing the next instruction. The unmasked processors set their elements of A to 1. At this point the mask bits are flipped, so that previously active processors become inactive, and vice versa. Now the unmasked processors set their elements of A to -1. Finally, all the mask bits are erased.

While the processor array is highly efficient when executing code in which every processor performs every operation, its efficiency can drop rapidly when the program enters conditionally executed code. First, there is the additional overhead of performing the tests to set the mask bits. Second, there is the inefficiency caused by having to work through different branches of control structures sequentially. For example, consider the case of a parallel **if-then-else** statement, where the conditional expression contains a parallel variable. First the **then** clause is executed by the processors for which the condition evaluated to true; the other processors are inactive. Next the active/inactive processors switch roles for the execution of the **else** clause. Overall, when the cost of evaluating the conditional expression is taken into account, the performance of the system performing an **if-then-else** statement is less than half the performance of the system performing parallel operations across the entire processor array.

2.3.5 Additional Architectural Features

A data path allows the front-end computer to access individual memory locations in the processor array. This capability is important, because it allows particular elements of parallel variables to be used or defined in sequential code. In this way, the processor array can be viewed as an extension of the memory space of the front-end computer.

A global result mechanism enables values from the processor array to be combined and returned to the front end. The ability to compute a global and is valuable. For example, some iterative programs may continue until all of the values in a matrix have converged. Suppose each element of the processor array contributes a 1 to the global result if its corresponding matrix element has converged, and a 0 if it has not converged. A global and of the values returned by the processors will return 1 if and only if all of the matrix elements have converged, making it easy to determine if the program should continue for another iteration.

2.3.6 Shortcomings of Processor Arrays

Processor arrays have several significant shortcomings that make them unattractive as general-purpose parallel computers.

First, many problems do not map well into a strict data-parallel solution. These problems cannot run efficiently on a processor array architecture.

Second, since the processor array can only execute a single instruction at a time, the efficiency of the computer drops when the program enters conditionally executed parallel code. Nested **if-then-else** statements or case statements are particularly bad.

Third, processor arrays are most naturally single-user systems. They do not easily accommodate multiple users who are simultaneously trying to execute multiple parallel programs. Dividing back-end processors into disjoint pools or dividing each processor's memory into multiple partitions requires sophisticated hardware and/or software enhancements to the computer.

Fourth, processor arrays do not scale down well. In order for a processor array with a large number of processors to exhibit high performance, the system needs high-bandwidth communication networks between the front end and the processor array, among the processing elements, and between the processing elements and parallel I/O devices. The cost of these networks may be only a small fraction of the cost of the system when the number of processing elements is high, but they may be a large fraction of the system cost when the number of processing elements is low. While the full-blown system may provide high performance per dollar, "introductory" systems with fewer processors do not exhibit good price/performance compared with competing systems.

Fifth, because processor arrays are built using custom VLSI, companies constructing processor arrays cannot "ride the wave" of performance and cost improvements manifested by commodity CPUs. Over time, it has proven to be increasingly difficult (or impossible) for companies producing custom processors to stay competitive with semiconductor manufacturers such as Intel, which can spend hundreds of millions of dollars on a new chip design and amortize this expense over the sale of millions of units.

Finally, one of the original motivations for constructing processor arrays—the relatively high cost of control units—is no longer valid. The amount of chip area dedicated to control circuitry is relatively small in today's CPUs.

For all these reasons, processor arrays are no longer considered a viable option for general-purpose parallel computers.

2.4 MULTIPROCESSORS

A **multiprocessor** is a multiple-CPU computer with a shared memory. The same address on two different CPUs refers to the same memory location. Multiprocessors avoid three of the problems associated with processor arrays. They can be built out of commodity CPUs, they naturally support multiple users, and they do not lose efficiency when encountering conditionally executed parallel code.

We discuss two fundamental types of multiprocessors: centralized multiprocessors, in which all the primary memory is in one place; and distributed multiprocessors, in which the primary memory is distributed among the processors.

2.4.1 Centralized Multiprocessors

A typical uniprocessor uses a bus to connect a CPU with primary memory and I/O processors. A cache memory helps keep the CPU busy by reducing the frequency at which the CPU must wait while instructions or data are fetched from primary memory.

A centralized multiprocessor is a straightforward extension of the uniprocessor. Additional CPUs are attached to the bus, and all the processors share the same primary memory (Figure 2.13). This architecture is also called a **uniform memory access (UMA) multiprocessor** or a **symmetric multiprocessor (SMP)**, because all the memory is in one place and has the same access time from every processor. Centralized multiprocessors are practical because the presence of large, efficient instruction and data caches reduces the load a single processor puts on the memory bus and memory, allowing these resources to be shared among multiple processors. Still, memory bus bandwidth typically limits to a few dozen the number of processors that can be profitably employed.

Private data are data items used only by a single processor, while **shared data** are data values used by multiple processors. In a centralized multiprocessor,

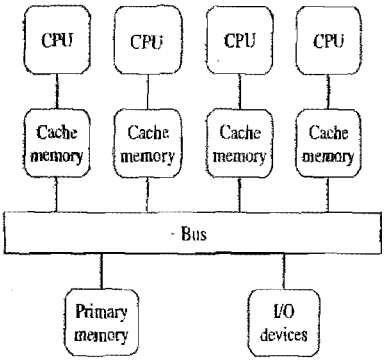


Figure 2.13 Architecture of a generic centralized multiprocessor.

processors communicate with each other through shared data values. For example, processors may be cooperating to perform all the tasks on a linked list. A shared pointer may contain the address of the next list item to be processed. Each processor accesses the shared pointer to determine its next task and advances the pointer before it is accessed by another processor. Designers of centralized multiprocessors must address two problems associated with shared data: the cache coherence problem and synchronization.

Cache Coherence Problem Replicating data across multiple caches reduces contention among processors for **shared data** values. However, because each processor's view of memory is through its cache, designers of systems with multiple processors must find a way to ensure that different processors do not have different values for the same memory location.

An example of how different processors can end up with different values appears in Figure 2.14. Two different CPUs, A and B, read the same memory location, and then CPU B writes a new value to that location. At this point CPU A has an obsolete image of that location's value still stored in its cache. This is called the **cache coherence problem.**

"Snooping" protocols are typically used to maintain cache coherence on centralized multiprocessors. Each CPU's cache controller monitors (snoops) the bus to identify which cache blocks are being requested by other CPUs. The most common solution to the cache coherence problem is to ensure that a processor has exclusive cache access to a data item before writing its value. Before the write occurs, all copies of the data item cached by other processors are invalidated. At this point the processor performs the write, updating the value in its cache block and in the appropriate memory location. When any other CPU tries to read a memory location from that cache block, it will experience a cache miss, forcing it to retrieve the updated value from memory. This is called the **write invalidate protocol.**

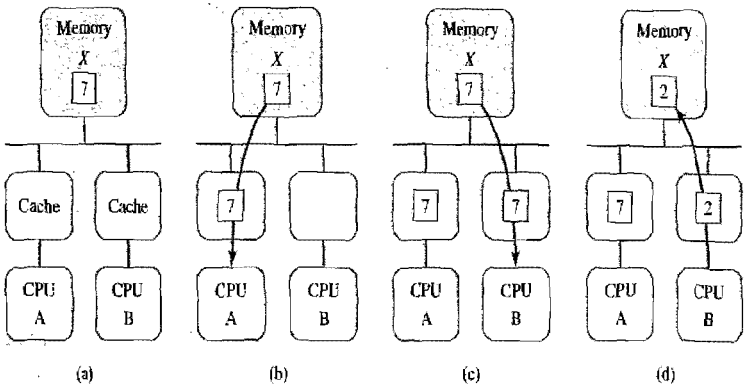


Figure 2.14 Example of the cache coherence problem. (a) Memory location *X* contains value 7. (b) CPU A reads *X*. A copy of *X* is stored in CPU A's cache. (b) CPU B reads *X*. A copy of *X* is stored in CPU B's cache. (c) CPU B stores 2 into *X*. Memory location *X* takes on the new value. The value is also updated in the cache of CPU B. However, CPU A still has the old value of *X* in its cache.

If two processors simultaneously try to write to the same memory location, only one of them wins the "race." The cache block of the "losing" processor is invalidated. The "losing" processor must get a new copy of the data (with the updated value) before it can do its write.

Processor Synchronization Various kinds of synchronization may be needed by processes cooperating to perform a computation. **Mutual exclusion** is "a situation in which at most one process can be engaged in a specified activity at any time" [116]. Earlier we gave the example of multiple processes cooperating to complete tasks stored on a linked list. Retrieving the next task from the list and updating the list pointer is an example of a situation demanding mutual exclusion.

Barrier synchronization is another kind of synchronization often found in shared-memory programs. A **barrier synchronization** guarantees that no process will proceed beyond a designated point in the program, called the barrier, until every process has reached the barrier. You might find a barrier synchronization between the two phases of a program's execution.

In most systems, software performing synchronization functions relies upon hardware-supported synchronization instructions. On systems with a small number of processors, the most common hardware synchronization mechanism is either an uninterruptible instruction or a sequence of instructions that atomically retrieve and change a value [90].

2.4.2 Distributed Multiprocessors

The existence of a shared memory bus limits to a few dozen the number of CPUs in a centralized multiprocessor. The alternative is to distribute primary memory

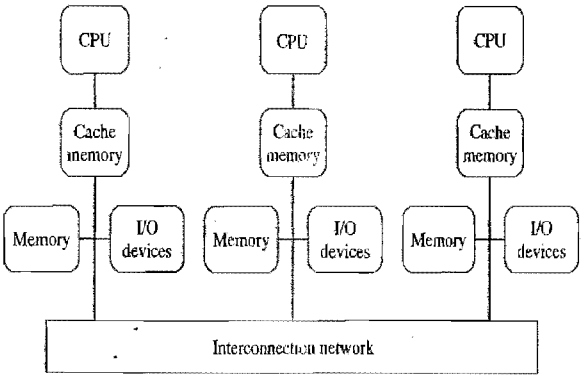


Figure 2.15 Architecture of a generic distributed-memory, multiple-CPU computer. If the computer has a single global address space, the computer is called a distributed multiprocessor. If the computer has disjoint local address spaces, it is called a multicomputer.

among the processors, creating a system in which local memory accesses are much faster than nonlocal memory accesses. Because executing programs exhibit spatial and temporal locality, it is possible to distribute instructions and data among memory units so that most of the system's memory references are between a processor and its local memory. Hence distributed memory systems can have higher aggregate memory bandwidth and lower memory access time, compared with centralized memory parallel computers. The net effect is to allow a higher processor count.

Distributing I/O, too, can also improve scalability. The architecture of a generic distributed-memory, multiple-CPU computer appears in Figure 2.15.

If the distributed collection of memories forms one logical address space, the parallel computer system is called a **distributed multiprocessor.** In a distributed multiprocessor, the same address on different processors refers to the same memory location. This type of system is also called a **nonuniform memory access (NUMA) multiprocessor,** because memory access time varies considerably, depending upon whether the address being referenced is in that processor's local memory or another processor's local memory.

Support for Cache Coherence Some distributed multiprocessors, such as the Cray T3D, do not have cache coherence hardware. On such computers, only instructions and private data can be stored in a processor's cache. This performance disadvantage is exacerbated by the huge time difference between a local cache access and a nonlocal memory access. For example, a nonlocal memory access takes 150 cycles on the Cray T3D, versus the two cycles needed for a cache reference. For these reasons, hardware support for cache coherence is valuable. Unfortunately, the snooping method described for centralized multiprocessors do not scale well as the number of processors grows, because a cache controller

cannot simply "snoop" on a shared memory bus. Instead, a more complicated protocol is needed.

Implementing a **directory-based protocol** is a popular way to implement cache coherence on a distributed multiprocessor. A single directory contains sharing information about every memory block that may be cached.

For each cache block, the directory entry indicates whether it is:

- uncached—not currently in any processor's cache
- shared—cached by one or more processors, and the copy in memory is correct
- exclusive—cached by exactly one processor that has written the block, so that the copy in memory is obsolete

It is necessary to keep track of which processors have copies of any cache block, so that these copies can be invalidated when one processor writes a value to that block. If the number of processors is 128 or less, it is reasonable to store information about which processors are sharing a data block as a bit vector.

To prevent accesses to the cache directory from becoming a performance bottleneck, the directory itself should be distributed among the computer's local memories. However, the contents are not replicated: the information about a particular memory block is in exactly one location.

Directory-Based Protocol Example Let's look at an example that illustrates how a directory-based protocol works. Consider the simple distributed memory multiprocessor shown in Figure 2.16a. The parallel computer has three CPUs. Associated with each processor is a cache, a memory, and a directory. Together, the memories of the individual processors form a single address space, and any CPU can reference any of these addresses. Integer variable X is stored in the memory controlled by processor 2. It currently has the value 7. Processor 2 has a directory entry corresponding to the cache block containing X. This entry shows that currently the block is uncached.

Now suppose CPU 0 tries to read the value of X. The cache block containing X is not in CPU 0's cache. A "read miss" message is sent from processor 0 to processor 2. The status of the cache block containing X is changed to "shared," the bit vector is updated to show that processor 0 has a copy of the cache block, and the block is sent to processor 0 (Figure 2.16b).

Next CPU 2 tries to read the value of X. The cache block containing X is not in CPU 2's cache. As a result of the read miss, the bit vector is changed to show that processor 2 also has a copy of the cache block, and the block is sent to processor 2's cache (Figure 2.16c).

Suppose CPU 0 now writes 6 to X. A "write miss" message is sent from processor 0 to processor 2. The directory controller invalidates the copy of the cache block currently in CPU 2's cache, updates the bit vector to show that CPU 2 no longer has a copy of the block, and changes the state of the cache block to "exclusive." Figure 2.16d shows the new state of the system. Note that the value of X in primary memory is out of date.

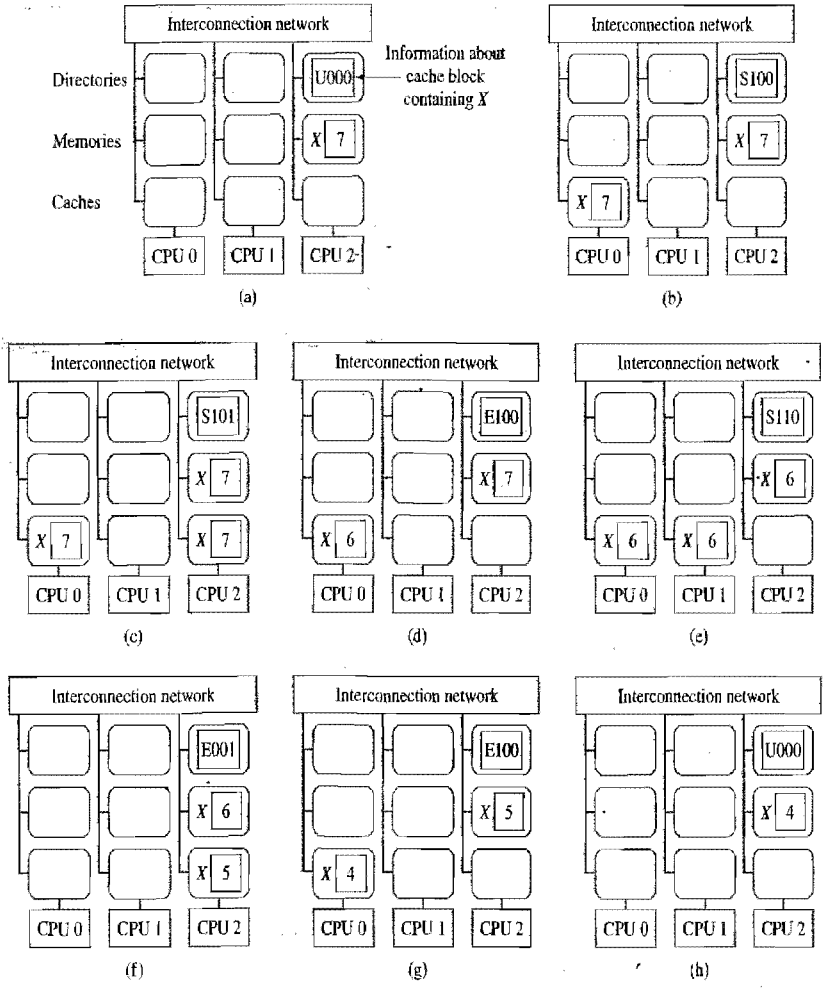


Figure 2.16 Illustration of a directory-based protocol to implement cache coherence on a distributed multiprocessor. (a) X has value 7. Block containing X is uncached. (b) State after CPU 0 reads X. (c) State after CPU 2 reads X. (d) State after CPU 0 writes value 6 to X. (e) State after CPU 1 reads X. (f) State after CPU 2 writes 5 to X. (g) State after CPU 0 writes 4 to X. (h) State after CPU 0 flushes cache block containing X.

If CPU 1 now tries to read X, generating a read miss, the directory controller for processor 2 sends a "switch to shared" message to processor 0. Processor 0 sends a copy of the cache block back to processor 2, so that an up-to-date copy is in primary memory. Then the up-to-date block is sent to processor 1 (Figure 2.16e).

Suppose the next action involving the cache block is CPU 2 writing 5 to X. Since the block is no longer in its cache, it generates a "write miss" message back

to the directory controller. The directory controller sends invalidate messages to processors 0 and 1, which remove the blocks from their caches. Now a status of the block is changed to "exclusive" with owner 2. A copy of the block is sent to CPU 2's cache, and CPU 2 updates the value of X (see Figure 2.16f).

Soon after, CPU 0 tries to write the value 4 to X. Since the appropriate block is not in its cache, processor 0 generates a "write miss" message to processor 2's directory. Processor 2 sends a "take away" message to CPU 2's cache controller. The cache block is copied back to memory. The state of the block remains exclusive, but the bit vector is updated to show the owner is now processor 0. A copy of the block is sent to CPU 0's cache, and the value of X is updated, as shown in Figure 2.16g.

Finally, suppose that processor 0 decides to flush the cache block containing X. Since it has exclusive access to the block, it must copy the contents of the block back to processor 2's memory. The final state is illustrated in Figure 2.16h.

2.5 MULTICOMPUTERS

A **multicomputer** is another example of a distributed-memory, multiple-CPU computer, as illustrated in Figure 2.15. However, unlike a NUMA multiprocessor, which has a single global address space, a multicomputer has disjoint local address spaces. Each processor only has direct access to its own local memory. The same address on different processors refers to two different physical memory locations. Without a shared address space, processors interact with each other by passing messages, and there are no cache coherence problems to solve.

Commercial multicomputers typically provide a custom switching network to provide low-latency, high-bandwidth access between processors. Commercial systems usually provide a good balance between the speed of the processors and the speed of the communication network. In contrast, **commodity clusters** rely upon mass-produced computers, switches, and other equipment used to construct local area networks. This makes for a less expensive system, albeit one in which the message latency is higher and communication bandwidth is lower.

2.5.1 Asymmetrical Multicomputers

Early multicomputers often had an asymmetrical design (Figure 2.17), which in certain respects resembles a processor array. A front-end computer interacts with users and I/O devices, while the processors in the back end are dedicated to "number crunching." Two examples of multicomputers with an asymmetrical design are the Intel iPSC and nCUBE/ten. The Intel iPSC (c. 1984) consisted of a Cube Manager (front end) controlling up to 128 processing nodes. The processing nodes ran the NX operating system. The nCUBE/ten (c. 1985) consisted of an Intel 80286 host processor (front end) running AXIS, a custom multiprogrammed operating system, controlling up to 1024 nodes running the extremely small (4 Kbyte) VERTEX operating system.

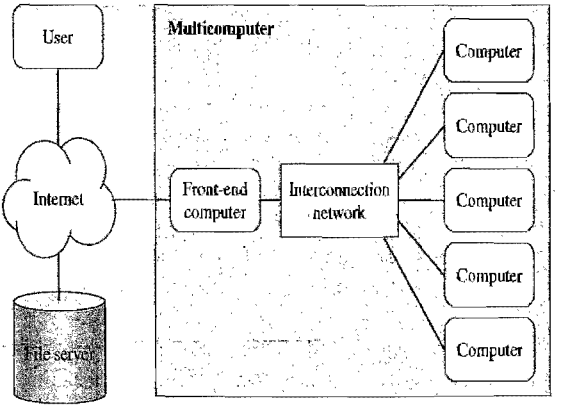


Figure 2.17 An asymmetrical multicomputer. Users log into the front-end computer, which executes a full, multiprogrammed operating system and provides all functions needed for program development. The computers in the back end are reserved for executing parallel programs. These computers may execute a primitive operating system. In a variant of this design, multiple computers may constitute the front end.

In an **asymmetrical multicomputer,** back-end processors are used exclusively for executing parallel programs. These processors may be running a primitive operating system, such as VERTEX, that does not support virtual memory, I/O, or even multiprogramming. With no other processes occupying CPU cycles or sending messages across the network, it is easier to understand, model, and tune the performance of a parallel application. This is a significant advantage. Another advantage, which was important in the early days of commercial multicomputers, is the ease with which a manufacturer can develop the primitive operating system needed for the back-end processors.

Asymmetrical multicomputers have several significant disadvantages. First, there is a single point of failure. If the front-end computer is "down," the entire parallel computer is out of action.

Second, scalability is limited by the performance capabilities of the front-end computer. Users log into the front end and use it for program editing and compiling. The front end launches parallel programs running on the back end. It is also responsible for any I/O operations. As the number of users increases, the front end may become overloaded. Meanwhile, a significant portion of the processors in the back end may be idle.

Introducing multiple front-end computers provides additional computing resources where they may be needed, but results in additional complexities. For example, how do users know which front-end computer to log into? How will the workload be balanced between the front-end computers? Will back-end nodes be assigned statically or dynamically to particular front-end processors?

Another solution to the front-end performance bottleneck problem is to improve the performance of a single front-end computer, perhaps by replacing a single-CPU system with a centralized multiprocessor. Of course, the existence of an underutilized multiprocessor front end could frustrate users who would like to use some of its computing capabilities in a parallel computation.

A third disadvantage of asymmetrical multicomputers has to do with program debugging. Primitive operating systems in back-end processors may make program performance easier to understand, but they also make debugging programs much more difficult. Since they do not support I/O operations, it is impossible for a node program to print a message to the user. Instead, the node program must send a message to the front-end computer, which then can pass the message along to the user by printing its contents.

This leads us to a fourth disadvantage of asymmetrical multicomputers. Every parallel application requires the development of two distinct programs: the front-end program and the back-end program. The front-end program is responsible for interacting with the user and the file system, transmitting data to the back-end processors, and forwarding results from the back-end processors to the outside world. The back-end program is responsible for implementing the computationally intensive portion of the algorithm. Developing two programs for every application is tedious and error prone.

The difficulty of debugging parallel programs is a strong incentive to provide full-featured I/O facilities on back-end nodes. A straightforward way to do this is to run a multiprogrammed operating system on the back-end processors, too. At this point, the difference between the front end and the back end is down to which nodes users can log into. If the front-end computers have excessive loads and the back-end computers are underutilized, there is a strong incentive to open every computer up to program development.

2.5.2 Symmetrical Multicomputers

In a **symmetrical multicomputer,** every computer executes the same operating system and has identical functionality. Users may log into any computer to edit and compile their programs. Any or all of the computers may be involved in the execution of a particular parallel program (Figure 2.18).

Symmetrical multicomputers can solve many of the problems encountered in asymmetrical multicomputers. For example, they alleviate the performance bottleneck caused when a single computer serves as the site for program development. If one computer has a heavy load, users can log into another.

Support for debugging is better in symmetrical multicomputers. Since every computer runs a full-fledged operating system, every processor can write a debugging message back to the user.

Symmetrical multicomputers also eliminate the "front-end/back-end" programming problem. Every processor executes the same program. When only one processor should perform a particular operation, it is easily selected with an **if** statement.

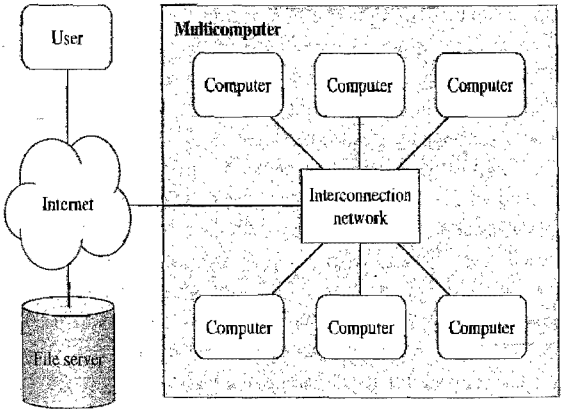


Figure 2.18 A symmetrical multicomputer. Every computer supports a full, multiprogrammed operating system. Users may log into any of these computers. Any or all of the computers may be called upon to execute a parallel program.

Symmetrical multicomputers have disadvantages, too. First, it is more difficult to maintain the illusion of a single "parallel computer" when a user can log into any node in the system, each with its own name.

Second, there is no simple way to balance the program development workload among all the processors. Even if a user checks the loads on the computers before logging in, these loads change with time. Without software support, it is likely that the system's workload will be unbalanced.

Third, it is more difficult to achieve high performance from parallel programs when processes must compete with other processes for CPU cycles, cache space, and memory bandwidth. Cache memory is processor-oriented, not process-oriented. A context switch from one process to another often results in a large number of cache misses, lowering performance.

2.5.3 Which Model Is Best for a Commodity Cluster?

For ease of programming and debugging, it makes sense to put a full-fledged multiprogrammed operating system (such as Linux) on every computer and give every computer access to the file system. Symmetrical multicomputers have this characteristic.

The performance of a CPU on a given application depends to a large degree on its cache hit rate. If the primary goal of the system is to maximize the performance of individual parallel programs, it is a good idea to put only a single user process on each CPU. This is an argument for an asymmetrical arrangement in which most of the nodes are off-limits to program development.

Parallel program performance can also be limited by the speed of the network. For this reason it makes sense to give only parallel processes access to the

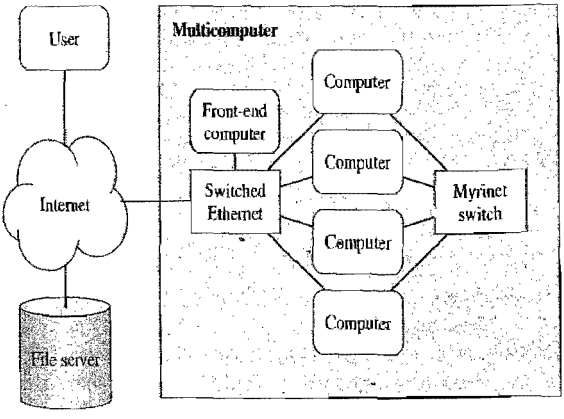


Figure 2.19 The ParPar cluster mixes features from the asymmetrical and symmetrical models.

interprocessor connection network. Users should access the front-end computer by another route.

In short, the best arrangement for a commodity cluster may well be a mixed model that has attributes of both the asymmetrical and symmetrical designs. Figure 2.19 illustrates the ParPar cluster at the Hebrew University of Jerusalem [24]. The front-end computer and 16 back-end computers are all connected to the Internet via switched Ethernet. However, the back-end computers also have exclusive access to a high-speed dedicated data network.

2.5.4 Differences between Clusters and Networks of Workstations

A commodity cluster contains components found in local area networks—commodity computers and switches. You can execute parallel programs on both clusters and local area networks. What sort of system ought to be called a cluster?

A network of workstations is a dispersed collection of computers, typically located on users' desks. Often the workstations are connected via Ethernet (10 Mbit/sec) or fast Ethernet (100 Mbit/sec). The principal role of a workstation is to serve the needs of the person using it; executing parallel jobs is simply a way to consume leftover CPU cycles. Individual workstations may have different operating systems and executable programs. Users have the power to turn off their workstations. For this reason, there is a greater need to support checkpointing and restarting of jobs.

In contrast, a commodity cluster is usually a co-located collection of mass-produced computers and switches dedicated to running parallel jobs. There is a good chance the computers are accessible only via the network; in other words, the computers typically do not have displays or keyboards. Some of the computers may not allow users to log in. All of the computers run the same version of the

Table 2.2 Comparison of three options for switched networks in commodity clusters (circa 2002). Cost per node includes the price of the network interface card and one node's share of the switch price.

	Latency	Bandwidth	Cost/node
Fast Ethernet	100 μsec	100 Mbit/sec	<$100
Gigabit Ethernet	100 μsec	1000 Mbit/sec	<$1,000
Myrinet	7 μsec	1920 Mbit/sec	<$2,000

same operating system and have identical local disk images. The entire cluster is administered as an entity.

Another key distinction between a commodity cluster and a network of workstations is the speed of the network. Given the speed of today's computers, Ethernet is simply too slow to be used as the network underlying a commodity cluster. In addition, it is essential that the networking medium be switched, not shared. (Switches, not hubs, must be used as connection devices.) Three popular switched networking options available to designers of commodity clusters are fast Ethernet, gigabit Ethernet, and Myrinet. Table 2.2 summarizes the differences between them.

2.6 FLYNN'S TAXONOMY

Flynn's taxonomy is the best-known classification scheme for parallel computers. In this scheme, a computer's category depends upon the parallelism it exhibits in its instruction stream and its data stream. A process can be seen as executing a sequence of instructions (the instruction stream) that manipulates a sequence of operands (the data stream). The focus is on the multiplicity of hardware used to manipulate the instruction and data streams [28, 29, 30].

A computer's hardware may support a single instruction stream or multiple instruction streams manipulating a single data stream or multiple data streams. Hence Flynn's classification results in four categories (Figure 2.20).

2.6.1 SISD

The category SISD refers to computers with a single instruction stream and a single data stream. Uniprocessors fall into this category. Even though it has only a single CPU executing a single instruction stream, a modern uniprocessor may still exhibit some concurrency of execution. For example, superscalar architectures support the dynamic identification and selection of multiple independent operations that may be executed simultaneously. Instruction prefetching and pipelined execution of instructions are other examples of concurrency typically found in modern SISD computers, though according to Flynn these are examples of concurrency of *processing*, rather than concurrency of *execution* [30].

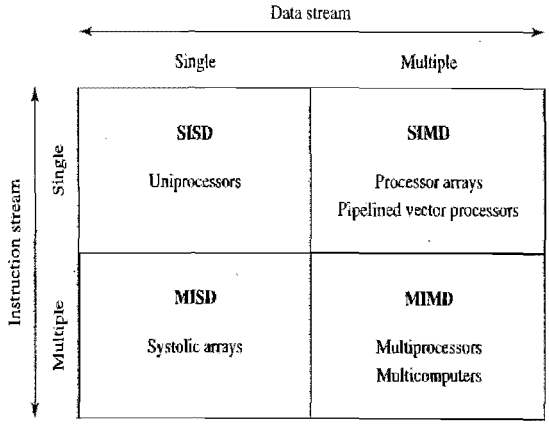


Figure 2.20 Flynn's taxonomy of computer architectures.

2.6.2 SIMD

The category SIMD refers to computers with a single instruction stream but multiple data streams. In this category are processor arrays and pipelined vector processors. As we have seen, a **processor array** is a parallel computer with a single control unit executing one instruction stream, as well as multiple subordinate processors capable of simultaneously performing the same operation on different data elements. A **pipelined vector processor** relies upon a very fast clock and one or more pipelined functional units to execute the same operation on the elements of a dataset.

2.6.3 MISD

The MISD category is for computers with multiple instruction streams, but only a single data stream. An MISD computer is "a pipeline of multiple independently executing functional units operating on a single stream of data, forwarding results from one functional unit to the next" [30].

A systolic array is an example of an MISD computer. The name comes from the word *systole*, which refers to a contraction of the heart. A **systolic array** is a network of primitive processing elements that "pump" data.

For example, consider the primitive sorting element of Figure 2.21. The sorter works in two phases. In the first phase (Figure 2.21a) it inputs three data values. In the second phase (Figure 2.21b) it outputs the minimum, median, and maximum values.

We can create a hardware priority queue by connecting a linear array of these sorting elements [70]. See Figure 2.22. The priority queue supports two

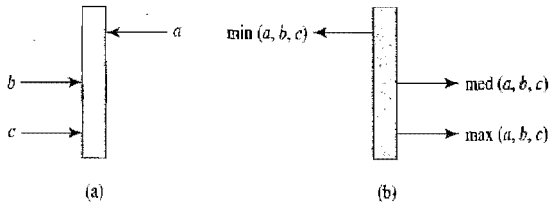


Figure 2.21 A systolic sorting element. (a) During the first cycle the element inputs keys *a*, *b*, and *c*. (b) During the second cycle the element outputs the minimum, median, and maximum of the three keys along designated channels.

operations: inserting a key and extracting the key with the minimum value. Each operation takes two cycles—that is, constant time.

To insert key x, the host processor inserts x and $-\infty$ into the left end of the priority queue during the first cycle. In the second cycle the queue outputs $-\infty$ at its left end. The host discards this value.

To extract the minimum key, the host inserts two copies of the ∞ key during the first cycle and extracts the minimum key during the second cycle. For all operations the key ∞ is inserted into the right end of the systolic array during the first clock cycle. In the second clock cycle, two copies of ∞ should be output from the right end of the systolic array. If one of the keys is not ∞, the priority queue has overflowed.

In this case all of the elements in the systolic array are identical. However, a systolic array can contain a variety of elements performing different functions, which is why it is appropriately called a "multiple instruction" architecture.

While a few commercial parallel systems based on MISD principles have been developed, they have been targeted to particular applications, such as digital signal processing. Flynn and Rudd suggest that the lack of a natural mapping from familiar programming constructs to the MISD organization has stifled interest in this architecture [30].

2.6.4 MIMD

The MIMD category is for computers with multiple instruction streams and multiple data streams. Multiprocessors and multicomputers fit into this category. Both of these architectures are based on multiple CPUs. Different CPUs can simultaneously execute different instruction streams manipulating different data streams.

Most contemporary parallel computers fall into Flynn's MIMD category. Hence the MIMD designation is not particularly helpful when describing modern parallel architectures. For the rest of the book we will rely upon the more specific terminology already developed in this chapter to describe the parallel architectures upon which our programs are executing.

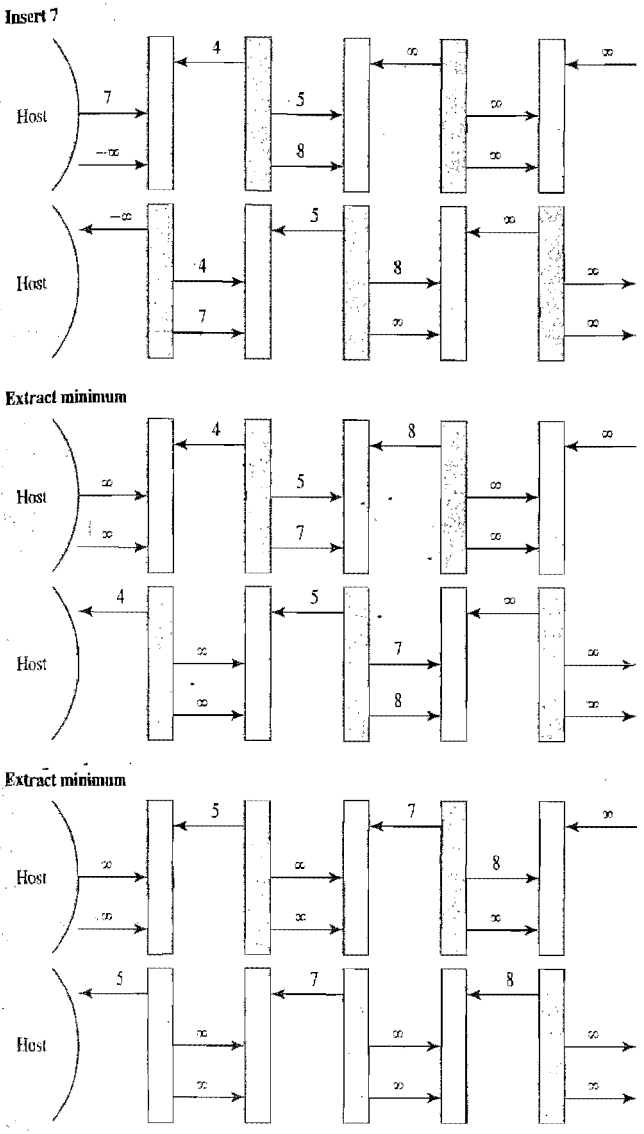


Figure 2.22 Operation of a systolic array implementing a priority queue in hardware.

2.7 SUMMARY

Since the mid-1960s scientists and engineers have designed and built a wide variety of parallel computers. Commercial parallel computers containing eight or more processors first began appearing in the early 1980s. With hindsight, we can appreciate why some architectures have been more successful than others as general-purpose computing platforms.

The construction of processor arrays was stimulated by the relatively high cost of control units and the preponderance of data-parallel operations in scientific computations. The first advantage of processor arrays, a single control unit, became insignificant when VLSI fabrication made possible CPUs on a chip. The second advantage, efficient execution of data-parallel operations, is diminished when these operations occur inside conditionally executed code.

Meanwhile, the many disadvantages of processor arrays became clear with time. Many problems are not amenable to a strict data-parallel solution. Processor arrays do not naturally support multiple users. They do not scale down well, making it difficult for them to compete with other systems at the low end of the market. The most significant disadvantage of processor arrays is that they are built using custom VLSI and cannot leverage the performance and cost improvements exhibited by commodity CPUs.

For these reasons processor arrays have receded into the shadows while multiple-CPU systems are getting the limelight. The vast majority of commercial parallel computers contain no more than a few dozen CPUs. These relatively small-scale systems are adequate for most high-performance computing needs because individual commodity CPUs are now quite powerful.

Most small-scale parallel computers have a centralized multiprocessor architecture. Typically processors access a common local memory through a shared bus. Architects of these systems must address the cache coherence problem and synchronization. Cache coherency is usually assured by implementing snoopy caches and using the write invalidation protocol to invalidate obsolete cache blocks whenever a write occurs. Software synchronization mechanisms rely upon one or more hardware instructions that have the net effect of atomically reading and updating a memory location.

For parallel computers containing 100 or more CPUs, some form of distributed memory is needed in order to provide sufficient memory bandwidth to the CPUs. In a distributed-memory system, each CPU has a nearby local memory. When most memory references are to cache memory or the nearest local memory, the aggregate memory bandwidth can be high, scaling with the number of processors in the computer.

Distributed-memory parallel computers are divided into two categories, depending upon whether they support a single global address space or have multiple disjoint address spaces. In a distributed multiprocessor, the same address on two different CPUs refers to the same memory location somewhere within the parallel computer. Cache coherence is more difficult to implement in a distributed multiprocessor, because there is no shared bus for cache controllers to snoop. Instead,

the most common way to implement cache coherence is through a directory-based scheme that uses bit vectors to record which CPUs have copies of which cache blocks.

A multicomputer is a distributed-memory parallel computer with multiple disjoint address spaces. The same address on two different CPUs refers to different memory locations. With no shared memory, there is no cache coherence problem to worry about. In order for processors to share data, they must send messages to each other. Whether or not a copy of a data item is up-to-date or not depends entirely upon the programmer. Commercial multicomputers typically have custom, high-performance interprocessor communication networks and message-passing software to ensure low-latency, high-bandwidth communications between processors.

A commodity cluster is a particular kind of multicomputer constructed out of mass-produced computers and networking devices. Commodity clusters can incorporate the latest commercial technology and take advantage of the attractive price/performance ratios of these products. Because commodity clusters often have faster CPUs and slower networks than commercial parallel computers, they typically are not as well balanced and are suitable for a narrower range of applications.

Flynn's taxonomy is one of the best-known ways of categorizing parallel computers. Unfortunately, most contemporary parallel computers fall into the same category (MIMD), marginalizing the utility of this descriptor.

2.8 KEY TERMS

barrier synchronization
binary n-cube
cache coherence problem and protocols (directory-based, write invalidate)
commodity cluster
2-D mesh network
distributed multiprocessor
Flynn's taxonomy (SISD, SIMD, MISD, MIMD)
interconnection media (shared, switched)
interconnection networks (2-D mesh, binary tree, butterfly, hypercube, hypertree, shuffle-exchange)
multicomputer (asymmetrical, symmetrical)
multiprocessor (UMA, NUMA, SMP)
mutual exclusion
network attributes (bisection width, diameter, edges/node, edge length)
network topology (direct, indirect)
perfect shuffle
performance
private data
ranks
shared data
systolic array
vector computer (pipelined vector processor, processor array)

2.9 BIBLIOGRAPHIC NOTES

Chapter 8 of Patterson and Hennessy's *Computer Architecture: A Quantitative Approach* is an excellent general introduction to multiprocessor architectures [90]. Nagendra and Rzymianowicz survey high-speed networks used to construct

commodity clusters [88]. Feitelson et al. have written an overview of ParPar, "a general-purpose, multi-user, MPP-like-system, using only off-the-shelf components" [24].

If you want to assemble your own commodity cluster, *Beowulf Cluster Computing with Linux* by Sterling et al. is a practical guide to the construction, management, and programming of commodity PC clusters running the Linux operating system [105].

2.10 EXERCISES

2.1 Draw hypercube networks with two, four, and eight nodes. Make sure you label the nodes. Is a hypercube network with n nodes a subgraph of a hypercube network with $2n$ nodes?

2.2 How many different ways can a d-dimensional hypercube be labeled?

2.3 The distance between nodes u and v in a graph is the length of the shortest path from u to v. Given a d-dimensional hypercube and a designated source node s, how many nodes are distance i from s, where $0 \leq i \leq d$?

2.4 Prove that if node u is distance i from node v in a hypercube, then there are $i!$ different paths of length i from u to v (though some hypercube edges may appear in more than one path).

2.5 Prove that if node u is distance i from node v in a hypercube, then there are i paths of length i from u to v that share no edges.

2.6 Prove that a hypercube has no cycles of odd length.

2.7 Give an algorithm that routes a message from node u to node v in an n-node hypercube in no more than $\log n$ steps.

2.8 Draw shuffle-exchange networks with two, four, and eight nodes. Make sure you label the nodes. Is a shuffle-exchange network with n nodes a subgraph of a shuffle-exchange network with $2n$ nodes?

2.9 Given a shuffle-exchange network with 2^k nodes, under what circumstances are nodes u and v exactly $2k - 1$ link traversals apart?

2.10 Give an algorithm that routes a message from node u to node v in an n-node shuffle-exchange network in no more than $2 \log n - 1$ messages.

2.11 An **omega network** is an indirect topology based upon the perfect shuffle interconnection pattern [66]. Figure 2.23 illustrates an omega network for eight processors. Consider an omega network connecting $n = 2^k$ processors.

a. How many switching elements are in the network?

b. What is the diameter of the network?

c. What is the bisection width of the network?

d. What is the maximum number of edges per switching node?

e. Does the network have constant edge length as the number of nodes increases?

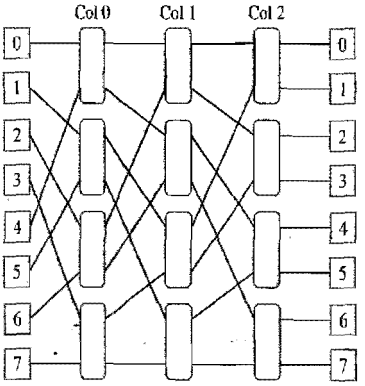


Figure 2.23 An omega network connecting eight processors, represented by squares.

2.12 Assume $n = 2^k$ processors are connected by an omega network (Figure 2.23). Design an algorithm to route a message from processor i to processor j. (Hint: Represent the destination address j as a binary number.)

2.13 Why are processor arrays well suited for executing data-parallel programs?

2.14 Given a processor array containing eight processing elements, each capable of performing 10 million integer operations per second, determine the performance in millions of operations per second of this processor array adding two integer vectors, for all vector sizes from 1 to 50.

2.15 Estimate the efficiency of a processor array executing a case statement with k cases. Assume all the instructions inside the case statement are parallel instructions, and assume all instructions take the same amount of time to execute.

a. What is the efficiency if each case contains the same number of instructions?

b. What is the efficiency if case i has I_i instructions and the probability of a processing element being active inside case i is P_i?

2.16 Why are large data and instruction caches desirable in multiprocessors?

2.17 Why is the number of processors in a centralized multiprocessor limited to a few dozen?

2.18 A directory-based protocol is a popular way to implement cache coherence on a distributed multiprocessor.

a. Why should the directory be distributed among the multiprocessor's local memories?

b. Why are the contents of the directory not replicated?

2.19 Continue the illustration of a directory-based cache coherence protocol begun in Figure 2.16. Assume the following five operations now occur in the order listed: CPU 2 reads X, CPU 2 write 5 to X, CPU 1 reads X, CPU 0 reads X, CPU 1 writes 9 to X. Show the states of the directories, caches, and memories after each of these operations.

2.20 Do some research and find, for each category in Flynn's taxonomy, at least one commercial computer fitting that category. (It is OK to name a computer that is no longer available, but you may not name a computer mentioned in this book.)

2.21 Continue the example of the operation of a systolic priority queue begun in Figure 2.22 by illustrating the states it would pass through as it processed these five requests: Insert 4, Extract Minimum, Insert 11, Insert 9, Extract Minimum.

2.22 Explain why contemporary supercomputers are invariably multicomputers.

CHAPTER

3

Parallel Algorithm Design

From the highest to the humblest tasks,
all are of equal honor; all have their part to play.
Winston Churchill

3.1 INTRODUCTION

It's time to start designing parallel algorithms! Our methodology is based on the task/channel model described by Ian Foster [31]. This model facilitates the development of efficient parallel programs, particularly those running on distributed-memory parallel computers.

The first two sections of this chapter describe the task/channel model and the fundamental steps of designing parallel algorithms based on this model. We then study a few simple problems. For each of these problems we design a task/channel parallel algorithm and derive an expression for its expected execution time. In the process our execution time model becomes increasingly sophisticated.

3.2 THE TASK/CHANNEL MODEL

The task/channel model represents a parallel computation as a set of tasks that may interact with each other by sending messages through channels (Figure 3.1). A **task** is a program, its local memory, and a collection of I/O ports. The local memory contains the program's instructions and its private data. A task can send local data values to other tasks via output ports. Conversely, a task can receive data values from other tasks via input ports.

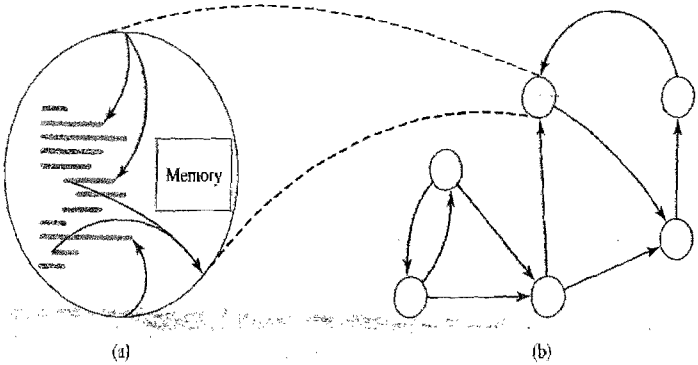


Figure 3.1 The task/channel programming model. (a) A task consists of a program, local memory, and a collection of I/O ports. (b) A parallel computation can be viewed as a directed graph in which vertices represent tasks and directed edges represent communication channels.

A **channel** is a message queue that connects one task's output port w another task's input port. Data values appear at the input port in the same orc in which they were placed in the output port at the other end of the channel.

Obviously, a task cannot receive a data value until the task at the other e of the channel has sent it. If a task tries to receive a value at an input port a no value is available, the task must wait until the value appears, and we say t receiving task has **blocked**. In contrast, a process sending a message never block even if previous messages it has sent along the same channel have not yet be received. Put another way, in the task/channel model receiving is a **synchrono** operation, while sending is an **asynchronous** operation.

In the task/channel model local accesses of private data are easily disti guished from nonlocal data accesses that occur over channels. This is goo because we should think of local accesses as being much faster than nonloc data accesses.

When we talk about the execution time of a parallel algorithm, we are referri to the period of time during which any task is active. The starting time is whe all tasks simultaneously begin executing. The finishing time is when the last tas has stopped executing.

3.3 FOSTER'S DESIGN METHODOLOGY

Ian Foster has proposed a four-step process for designing parallel algorithm [31]. It encourages the development of scalable parallel algorithms by delayin machine-dependent considerations to the later steps. We'll use Foster's desig methodology in this chapter and throughout the rest of the book to develop paralle algorithms for a wide variety of applications.

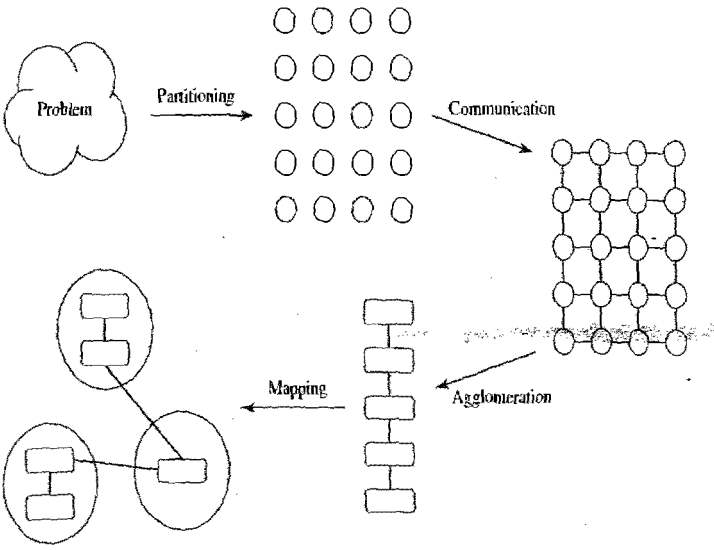


Figure 3.2 Foster's parallel algorithm design methodology.

The four design steps are called partitioning, communication, agglomeration, and mapping (Figure 3.2). In this section we explain each of these steps and provide a checklist that can help you determine if you're producing a good design. While the explanations at this point are rather theoretical, we'll spend the rest of the chapter grounding the theory by working through several practical examples.

3.3.1 Partitioning

When we begin the design of a parallel algorithm, we typically try to discover as much parallelism as possible. **Partitioning** is the process of dividing the computation and the data into pieces. A good partitioning splits both the computation and the data into many small pieces. To do this, we can either take a data-centric approach or a computation-centric approach.

Domain decomposition is the parallel algorithm design approach in which we first divide the data into pieces and then determine how to associate computations with the data. Typically our focus is on the largest and/or most frequently accessed data structure in the program.

Consider the example illustrated in Figure 3.3. Here a three-dimensional matrix is the largest and most frequently accessed data structure. We could partition the matrix into a collection of two-dimensional slices, resulting in a one-dimensional collection of primitive tasks. Alternatively, we could partition the matrix into a collection of one-dimensional slices, resulting in a two-dimensional collection of primitive tasks. Finally, we could consider each matrix element individually, producing a three-dimensional collection of primitive tasks. At this

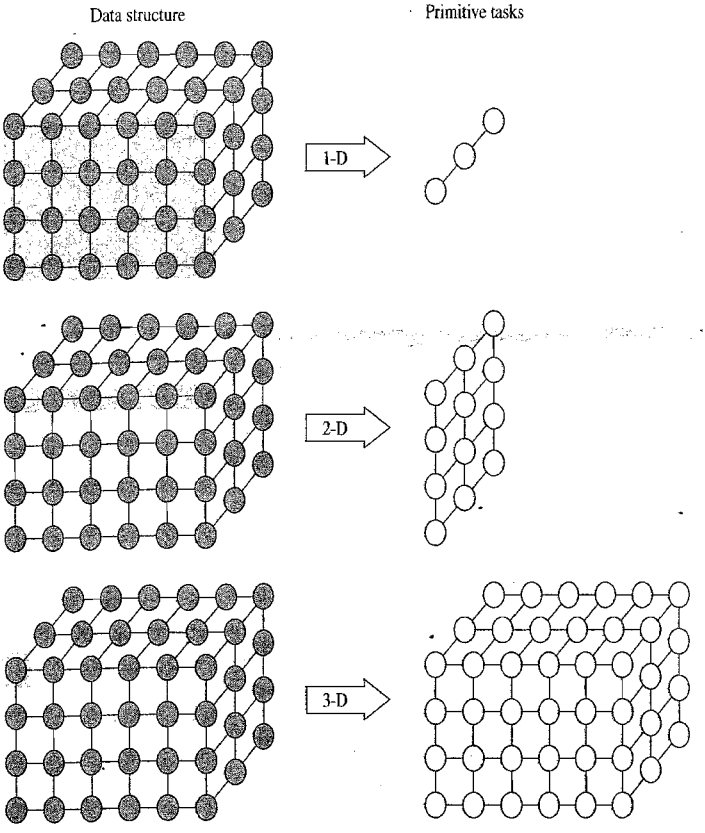


Figure 3.3 Three domain decompositions of a three-dimensional matrix, resulting in markedly different collections of primitive tasks.

point in the design process it is usually best to maximize the number of primitive tasks. Hence the three-dimensional partitioning is preferred.

Functional decomposition is the complementary strategy in which we first divide the computation into pieces and then determine how to associate data items with the individual computations. Often functional decompositions yield collections of tasks that achieve concurrency through pipelining.

For example, consider a high-performance system supporting interactive image-guided brain surgery (Figure 3.4) [37]. Before the surgery begins, the system inputs a set of CT scans of a patient's brain and registers these images, constructing a three-dimensional model. During surgery, the system tracks the position of the surgical instruments, converts them from physical coordinates to image coordinates, and displays on a monitor the position of the instruments amid the surrounding tissue. The system has inherent concurrency. While one task is converting an image from physical coordinates to image coordinates, a

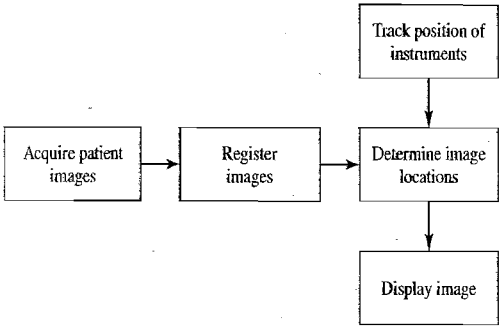


Figure 3.4 Functional decomposition of a system supporting interactive image-guided surgery.

second task can be displaying the previous image, and a third task can be tracking instrument positions for the next image.

Whichever decomposition we choose, we call each of these pieces a **primitive task.** Our goal is to identify as many primitive tasks as possible, because the number of primitive tasks is an upper bound on the parallelism we can exploit.

We can use the following checklist to evaluate the quality of a partitioning. The best designs satisfy all of these attributes (Foster [31]).

- There are at least an order of magnitude more primitive tasks than processors in the target parallel computer. (If this condition is not satisfied, later design options may be too constrained.)
- Redundant computations and redundant data structure storage are minimized. (If this condition is not satisfied, the design may not work well when the size of the problem increases.)
- Primitive tasks are roughly the same size. (If not, it may be hard to balance work among the processors.)
- The number of tasks is an increasing function of the problem size. (If not, it may be impossible to use more processors to solve larger problem instances.)

3.3.2 Communication

After we have identified the primitive tasks, the next step is to determine the communication pattern between them. Parallel algorithms have two kinds of communication patterns: local and global. When a task needs values from a small number of other tasks in order to perform a computation, we create channels from the tasks supplying the data to the task consuming the data. This is an example of a **local communication.**

In contrast, a **global communication** exists when a significant number of the primitive tasks must contribute data in order to perform a computation. An

example of a global communication is computing the sum of values held by the primitive processes. While it is important to note when global communications are needed, it is generally not helpful to draw communication channels for them at this stage of the algorithm's design.

We call communication among tasks part of the overhead of a parallel algorithm, because it is something the sequential algorithm does not need to do. Minimizing parallel overhead is an important goal of parallel algorithm design. Keeping this in mind, we can use Foster's checklist to help us evaluate the communication structure of our parallel algorithm.

- The communication operations are balanced among the tasks.
- Each task communicates with only a small number of neighbors.
- Tasks can perform their communications concurrently.
- Tasks can perform their computations concurrently.

3.3.3 Agglomeration

During the first two steps of the parallel algorithm design process, our focus was on identifying as much parallelism as possible. At this point we most likely do not have a design that would execute efficiently on a real parallel computer. For example, if the number of tasks exceeds the number of processors by several orders of magnitude, simply creating these tasks would be a source of significant overhead. In the final two steps of the design process we have a target architecture in mind (e.g., centralized multiprocessor or multicomputer). We consider how to combine primitive tasks into larger tasks and map them onto physical processors to reduce the amount of parallel overhead.

Agglomeration is the process of grouping tasks into larger tasks in order to improve performance or simplify programming. Sometimes we want the number of consolidated tasks to be greater than the number of processors on which our parallel algorithm will execute. Often, however, when developing MPI programs, we leave the agglomeration step with one task per processor. In this case, the mapping of tasks to processors is trivial.

One of the goals of agglomeration is to lower communication overhead. If we agglomerate primitive tasks that communicate with each other, then the communication is completely eliminated, because the data values controlled by the primitive tasks are now in the memory of the consolidated task (Figure 3.5a). We call this **increasing the locality** of the parallel algorithm. If the tasks cannot perform their computations concurrently, because later tasks are waiting for data provided by earlier tasks, then it's usually a good idea to agglomerate the tasks.

Another way to lower communication overhead is to combine groups of sending and receiving tasks, reducing the number of messages being sent (Figure 3.5b). Sending fewer, longer messages takes less time than sending more, shorter messages with the same total length because there is a message startup cost (called the message latency) incurred every time a message is sent, and this time is independent of the length of the message.

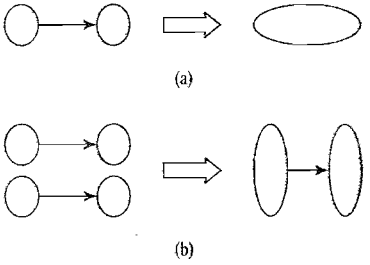


Figure 3.5 Agglomerating tasks can eliminate communications or at least reduce their overhead. (a) Combining tasks that are connected by a channel eliminates that communication, increasing the locality of the parallel algorithm. (b) Combining sending and receiving tasks reduces the number of message transmissions.

A second goal of agglomeration is to maintain the scalability of the parallel design. We want to ensure that we have not combined so many tasks that we will not be able to port our program at some point in the future to a computer with more processors. For example, suppose we are developing a parallel program that manipulates a three-dimensional matrix of size $8 \times 128 \times 256$. We plan to execute our program on a centralized multiprocessor with four CPUs. If we design the parallel algorithm so that the second and third dimensions are agglomerated, we could certainly execute the resulting program on four CPUs. Each task would be responsible for a $2 \times 128 \times 256$ submatrix. Without changing the design, we could even execute on a system with eight CPUs. Each task would be responsible for a $1 \times 128 \times 256$ submatrix. However, we could not port the program to a parallel computer with more than eight CPUs without changing the design, which would probably result in massive changes to the parallel code. Hence the decision to agglomerate the second and third dimensions of the matrix could turn out to be a shortsighted one.

A third goal of agglomeration is to reduce software engineering costs. If we are parallelizing a sequential program, one agglomeration may allow us to make greater use of the existing sequential code, reducing the time and expense of developing the parallel program.

We can use Foster's checklist to evaluate the quality of an agglomeration:

- The agglomeration has increased the locality of the parallel algorithm.
- Replicated computations take less time than the communications they replace.
- The amount of replicated data is small enough to allow the algorithm to scale.

- Agglomerated tasks have similar computational and communications costs.
- The number of tasks is an increasing function of the problem size.
- The number of tasks is as small as possible, yet at least as great as the number of processors in the likely target computers.
- The trade-off between the chosen agglomeration and the cost of modifications to existing sequential code is reasonable.

3.3.4 Mapping

Mapping is the process of assigning tasks to processors. If we are executing our program on a centralized multiprocessor, the operating system automatically maps processes to processors. Hence our discussion assumes the target system is a distributed-memory parallel computer.

The goals of mapping are to maximize processor utilization and minimize interprocessor communication. **Processor utilization** is the average percentage of time the system's processors are actively executing tasks necessary for the solution of the problem. Processor utilization is maximized when the computation is balanced evenly, allowing all processors to begin and end execution at the same time. (Conversely, processor utilization drops when one or more processors are idle while the remainder of the processors are still busy.)

Interprocessor communication increases when two tasks connected by a channel are mapped to different processors. Interprocessor communication decreases when two tasks connected by a channel are mapped to the same processor.

For example, consider the mapping of Figure 3.6. Eight tasks are mapped onto three processors. The left and right processors are responsible for two tasks, while the middle processor is responsible for four tasks. If all processors have the same speed and every task requires the same amount of time to be performed, then

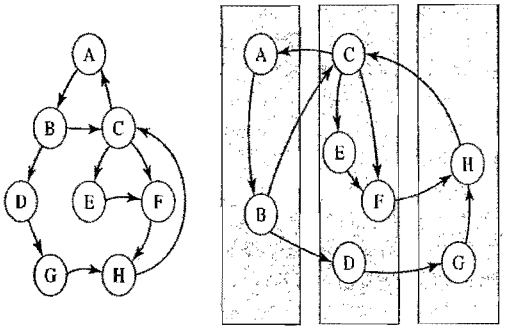


Figure 3.6 The mapping process. (a) A task/channel graph. (b) Mapping of tasks to three processors. Some channels now represent intraprocessor communications, while others represent interprocessor communications.

the middle processor will spend twice as much time executing tasks as the other two processors. If every channel communicates the same amount of data, then the middle processor will also be responsible for twice as many interprocessor communications as the other two processors.

Increasing processor utilization and minimizing interprocessor communication are often conflicting goals.

For example, suppose there are p processors available. Mapping every task to the same processor reduces interprocessor communication to zero, but reduces utilization to $1/p$. Our goal, then, is to choose a mapping that represents a reasonable middle point between maximizing utilization and minimizing communication.

Unfortunately, finding an optimal solution to the mapping problem is NP-hard [38], meaning there are no known polynomial-time algorithms to map tasks to processors to minimize the execution time. Hence we must rely on heuristics that can do a reasonably good job of mapping.

When a problem is partitioned using domain decomposition, the tasks remaining after the agglomeration step often have very similar size, meaning the computational loads are balanced among the tasks. If the communication pattern among the tasks is regular, a good strategy is to create p agglomerated tasks that minimize communication and map each of these tasks to its own processor.

Sometimes the number of tasks is fixed and the communication pattern among them is regular, but the time required to perform each task has significant variability. If nearby tasks tend to have similar computational requirements, then a cyclic (or interleaved) mapping of tasks to processors can result in a balanced computational load, at the expense of higher communications costs.

Some problems yield an unstructured communication pattern among the tasks. In this case it is important to map tasks to processors to minimize the communication overhead of the parallel program. A static load-balancing algorithm, executed before the program begins running, can determine the mapping strategy.

To this point, we have focused on designs utilizing a fixed number of tasks. Dynamic load-balancing algorithms are needed when tasks are created and destroyed at run-time or the communication or computational requirements of tasks vary widely. A dynamic load-balancing algorithm is invoked occasionally during the execution of the parallel program. It analyzes the current tasks and produces a new mapping of tasks to processors.

Finally, some parallel designs rely upon the creation of short-lived tasks to perform particular functions. Tasks do not communicate with each other. Instead, each task is given a subproblem to solve and returns with the solution to that subproblem. Task-scheduling algorithms can be centralized or distributed.

In a centralized task-scheduling algorithm, the pool of processors is divided into one manager and many workers. The manager processor maintains a list of tasks to be assigned. When a worker processor has nothing to do, it requests a task from the manager. The manager replies with a task. The worker completes the task, returns the solution, and requests another task. A potential problem with manager/worker–style task scheduling is that the manager can become a bottleneck. To some extent this problem can be ameliorated by allocating multiple tasks

at a time or allowing workers to prefetch tasks while they are working on earlier tasks.

In a distributed task-scheduling algorithm, each processor maintains its own list of available tasks. A mechanism is needed to spread the available tasks among the processors. Some algorithms rely on a "push" strategy. Processors with too many available tasks send some of them to neighboring processors. Other algorithms rely on a "pull" strategy. Processors with no work to do ask neighboring processors for work. A challenge with distributed task-scheduling algorithms is determining the termination condition. The uncompleted tasks are spread among the processors, and it is difficult for any process to know when all of them have been completed. In contrast, the manager process in a manager/worker–style algorithm always knows exactly how many uncompleted tasks remain.

Other task-scheduling algorithms represent a compromise between the centralized and decentralized algorithms we have described. For example, a two-level hierarchical manager/worker strategy has two levels of managers. The higher-level manager supervises a group of managers. Each lower-level manager allocates tasks to its own group of workers. Periodically the managers communicate with each other to balance the number of unassigned tasks held by each low-level manager.

Figure 3.7 summarizes how different characteristics of the parallel algorithm lead to different mapping strategies. Because the mapping strategy depends on

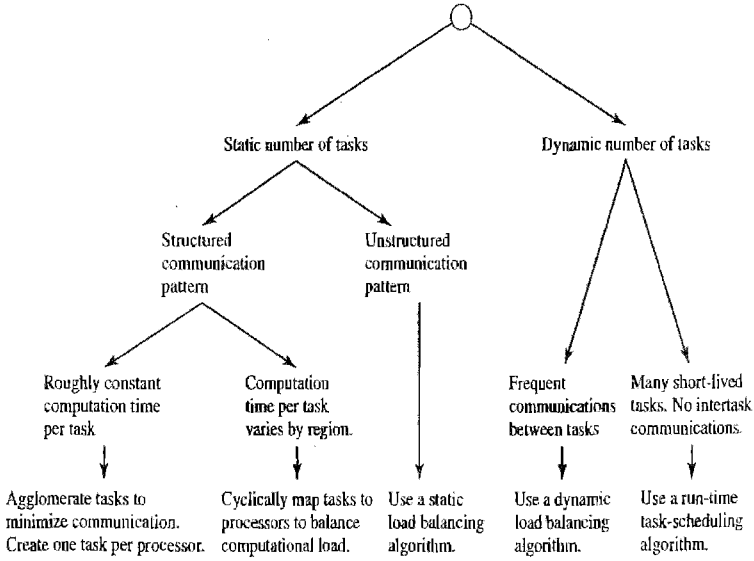


Figure 3.7 A decision tree to choose a mapping strategy. The best strategy depends on characteristics of the tasks produced as a result of the partitioning, communication, and agglomeration steps.

decisions made earlier in the parallel algorithm design process, it is important to keep an open mind during the design process. The following checklist (from Foster [31]) can help you decide if you've done a good job of considering design alternatives:

- Designs based on one task per processor and multiple tasks per processor have been considered.
- Both static and dynamic allocation of tasks to processors have been evaluated.
- If a dynamic allocation of tasks to processors has been chosen, the manager (task allocator) is not a bottleneck to performance.
- If a static allocation of tasks to processors has been chosen, the ratio of tasks to processors is at least 10:1.

3.4 BOUNDARY VALUE PROBLEM

3.4.1 Introduction

Let's apply our parallel algorithm design methodology to a simple, yet realistic, problem. See Figure 3.8. A thin rod made of uniform material is surrounded by a blanket of insulation so that temperature changes along the length of the rod are a result of heat transfer at the ends of the rod and heat conduction along the length of the rod. The rod has length 1. Both ends of the rod are exposed to an ice bath having temperature 0°C, while the initial temperature at distance x from the end of the rod is $100\sin(\pi x)$.

Over time, the rod gradually cools. A partial differential equation models the temperature at any point of the rod at any point in time. The finite difference method is one way to solve a partial differential equation on a computer. Figure 3.9 shows a finite difference approximation to the rod-cooling problem. Each curve represents the temperature distribution of the rod at some point in time. The curves drop as time increases. If you look carefully, you can see that each "curve" is actually composed of 10 line segments. In reality, the temperature distributions

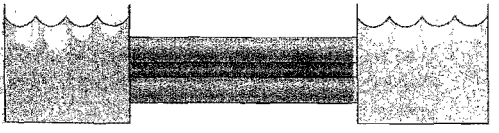


Figure 3.8 A thin rod (dark gray) is suspended between two ice baths. The ends of the rod are in contact with the icewater. The rod is surrounded by a thick blanket of insulation. We can use a partial differential equation to model the temperature at any point on the rod as a function of time.

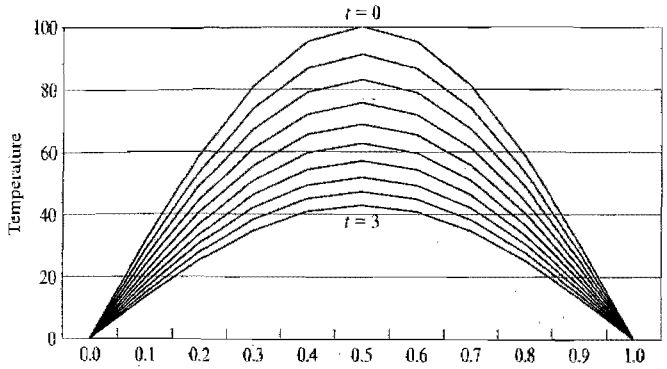


Figure 3.9 The rod cools as time progresses. The finite difference method finds the temperature at a fixed number of points in the rod at certain time intervals. Decreasing the size of the steps in space and time can lead to more accurate solutions.

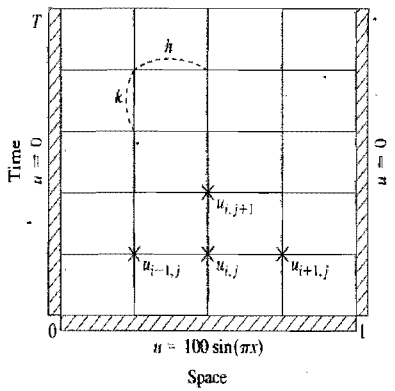


Figure 3.10 Data structure used in a finite difference approximation to the rod-cooling problem presented in Figure 3.8. Every point $u_{i,j}$ represents a matrix element containing the temperature at position i on the rod at time j. At each end of the rod the temperature is always 0. At time 0, the temperature at point x is $100 \sin(\pi x)$.

should be smooth curves. The finite difference method computes an approximate solution to the partial differential equation.

The finite difference method solving this problem stores temperatures in a two-dimensional matrix (Figure 3.10). Each row contains the temperature distribution of the rod at some point in time. The rod is divided into n sections of

length h, so each row has $n + 1$ elements. Increasing n reduces the error in the approximation. Time from 0 to T is divided into m discrete entities of length k, so the matrix contains $m + 1$ rows. The initial temperature distribution along the length of the rod is represented by the points in the bottom row. These values are known. The temperatures at the ends of the rod are represented by the left and right edges of the grid. These values, too, are known. Let $u_{i,j}$ represent the temperature of the rod at point i at time j.

In the finite difference method, the algorithm steps forward in time, using values from time j to compute the value for time $j + 1$ using the formula

$$u_{i,j+1} = ru_{i-1,j} + (1 - 2r)u_{i,j} + ru_{i+1,j}$$

where $r = k/h^2$.

3.4.2 Partitioning

Our first step is partitioning. In this case the data being manipulated are easy to identify: there is one data item per grid point. To start, let's associate one primitive task with each grid point. This yields a two-dimensional domain decomposition.

3.4.3 Communication

Now that we have identified our tasks, we need to determine the communication pattern between the tasks. If task A needs a value from task B to perform its computation, we need to draw a channel from task B to task A. Since the task computing $u_{i,j+1}$ requires the values of $u_{i-1,j}$, $u_{i,j}$, and $u_{i+1,j}$, in general each task will have three incoming channels and three outgoing channels (see Figure 3.11a). The tasks on the edges have fewer channels.

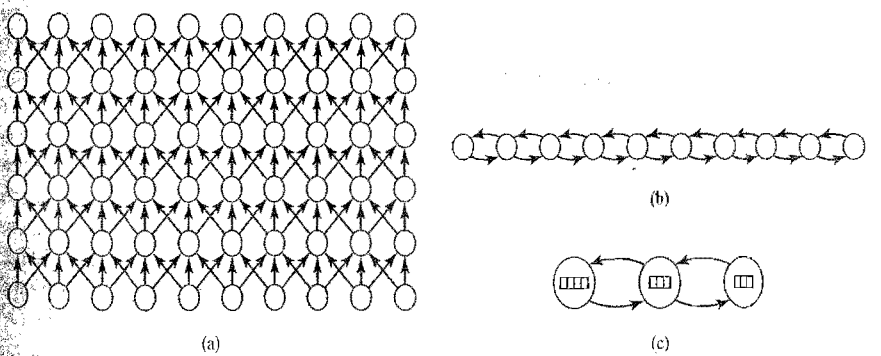


Figure 3.11 Task/channel graphs for parallel solutions to the boundary value problem. (a) The first domain decomposition associates one task with each temperature to be computed. (b) After the first agglomeration step, a single task now represents the computation of the temperature at element i for all time steps. (c) After the second agglomeration step, a task is responsible for computing, over all time steps, the temperatures for a contiguous group of rod locations.

3.4.4 Agglomeration and Mapping

Even if enough processors were available, it would be impossible to compute every task shown in Figure 3.11a concurrently, because the tasks computing rod temperatures later in time depend upon the results produced by tasks computing rod temperatures earlier in time. This is made plain by the vertical paths of channels stretching from the bottom tasks to the top tasks. There is no point in maintaining the illusion of multiple tasks when they must be performed sequentially. Let's agglomerate all the tasks associated with each point in the rod, that is, tasks in the same column in Figure 3.11a.

The resulting task/channel graph, shown in Figure 3.11b, is much less complicated. Now we have a linear array of tasks, each communicating solely with its neighbor(s). Each is responsible for computing the temperature at a particular grid point for all time steps.

However, even this graph is likely to have far more tasks than we need to keep all of our processors fully occupied, since in a real problem the number of rod segments would be large. We can use the decision tree of Figure 3.7 to come up with a mapping strategy. The number of tasks is static (left branch), the communication pattern among them is regular (left branch), and each task performs the same computations (left branch). Hence a good strategy is to create one task per processor, agglomerating primitive tasks so that computational workloads are balanced and communication is minimized. Associating a contiguous piece of the rod with each task (Figure 3.11c) preserves the simple nearest-neighbor communication between tasks and eliminates unnecessary communications for those data points within a single task.

3.4.5 Analysis

The rod has been divided into n pieces of size h. Let χ represent the time needed to compute $u_{i,j+1}$, given $u_{i-1,j}$, $u_{i,j}$, and $u_{i+1,j}$. Using a single processor to update the $n-1$ interior values of the rod requires time $(n-1)\chi$. Because the algorithm has m time steps, the total expected execution time of the sequential algorithm is $m(n-1)\chi$.

Now let's compute the expected execution time of the parallel algorithm. Let p denote the number of processors executing the algorithm. If each processor is responsible for an equal-sized portion of the rod's elements, the computation time for each iteration is $\chi\lceil(n-1)/p\rceil$. However, the parallel algorithm involves communication that the sequential algorithm does not, and we must account for that time. In general, each processor must send values to its two neighboring processors and receive two values from them. If λ represents the time needed for a processor to send (receive) a value to (from) another processor, then the necessary communications increase the parallel execution time for each iteration 2λ. *In our task/channel model a task may only send one message at a time, but it may receive a message at the same time it is sending a message.* Therefore, the task requires time 2λ to send data values to its two

neighbors but receives the two data values it needs from its neighbors at the same time.

Combining computation time with communication time, we see the overall parallel execution time per iteration is $\chi\lceil(n-1)/p\rceil + 2\lambda$, and our estimate of the parallel execution time for all m iterations of the algorithm is $m(\chi\lceil(n-1)/p\rceil + 2\lambda)$.

3.5 FINDING THE MAXIMUM

3.5.1 Introduction

The finite difference method we are using to compute the temperature distribution in the rod as a function of time only approximates the solution of the underlying partial differential equation. The reason we use finite difference or finite element methods to solve partial differential equations is that the boundary value problems arising from real-world situations are too complicated to solve analytically.

However, the heat conduction problem we examined in the previous section is simple enough to solve analytically. That means we can determine, for each of the m points along the rod, the difference between the computed solution and the correct solution. The error between the computed solution x and the correct solution c is $|(x-c)/c|$. Let's enhance our parallel algorithm to find the maximum error.

Given a set of n values $a_0, a_1, a_2, \ldots, a_{n-1}$ and an associative binary operator $\oplus$, **reduction** is the process of computing $a_0 \oplus a_1 \oplus a_2 \oplus \cdots \oplus a_{n-1}$. Addition is an example of an associative binary operator. Hence finding the sum $a_0 + a_1 + a_2 + \cdots + a_{n-1}$ is an example of a reduction.

You may not realize that minimum and maximum are associative binary operators, because they do not appear as operators in most programming languages. However, these two associative operators are extremely useful. For example, in the problem we are considering, we want to find the maximum value of a set.

Since reduction requires exactly $n-1$ operations, it has $\Theta(n)$ time complexity on a sequential computer. How quickly can we perform a reduction on a parallel computer? Without loss of generality, let's make the following explanation easier to read by assuming the operator is addition.

3.5.2 Partitioning

Since the list has n values, let's divide it into n pieces; in other words, as finely as possible. If we associate one task per piece, we have n tasks, each with one value. Our goal is to find the sum of all n values.

3.5.3 Communication

A task cannot directly access a value stored in the memory of another task. In order to compute the sum, we must set up channels between the tasks. A channel

from task A to task B allows task B to compute the sum of the values held by the two tasks. We want the communicating and summing to happen as quickly as possible. In one communication step each task may either send or receive one message.

At the end of the computation we want one task to have the grand total. We'll call this the root task. Let's start with a brute force approach: each of the other tasks sends its value to the root task, which adds up all the values (Figure 3.12a).

If it takes λ time for a task to communicate a value to another task and χ time to perform an addition, then this first parallel algorithm requires time

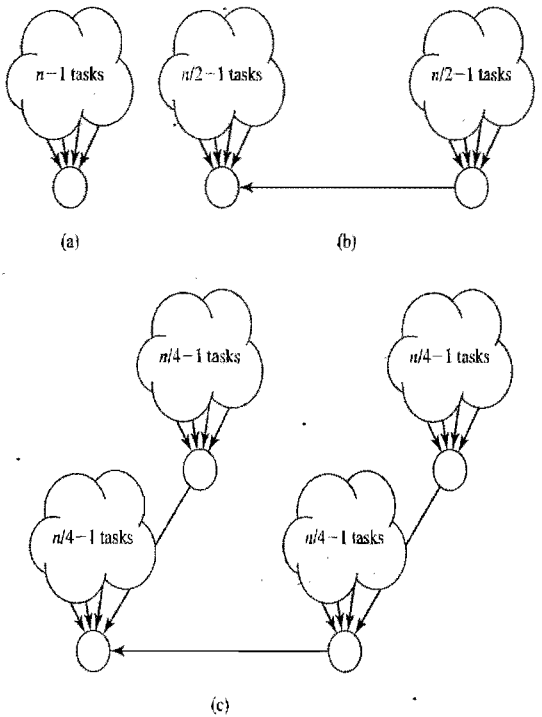


Figure 3.12 Evolution of an efficient parallel algorithm for reduction. (a) One task receives a list element from each of the other $n - 1$ tasks and performs all the additions. (b) Two tasks work together. Each receives list elements from $n/2 - 1$ other tasks. After $n/2$ addition steps, one task sends its subtotal to the first task. Compared to the original version, the computation time is cut nearly in half. (c) Four tasks cooperate. Each receives list elements from $n/4 - 1$ other tasks. After $n/4$ concurrent addition steps, there are four subtotals. These can be combined in two more communication/computation steps.

$(n-1)(\lambda+\chi)$. (The communication time is $(n-1)\lambda$ because the root task must receive $n-1$ messages.) This is actually slower than the sequential algorithm. We need to balance the communication and the computation better.

What if two tasks cooperated to perform the reduction? Let's have two semiroot tasks, each responsible for $n/2$ of the elements (Figure 3.12b). Now two communications can happen simultaneously, and after each communication two additions can happen at once. In time $(n/2-1)(\lambda+\chi)$ each semiroot task has a subtotal for its half of the elements. Now one of the semiroot tasks can pass its subtotal to the other task. In one additional communication/computation step a single task has the grand total. The expected execution time of this parallel algorithm is $(n/2)(\lambda+\chi)$.

Why not continue the process? What if we had four semiroot tasks, each responsible for $n/4$ of the list elements (Figure 3.12c)? We have increased the communication and computation concurrency to four. After the four subtotals have been computed, two remaining communication/computation steps yield the grand total. This algorithm is nearly four times as fast as the original algorithm.

If we take this notion to the limit, we have $n/2$ semiroot tasks, each responsible for two list elements. In the first step of the algorithm half the tasks send messages to the other half of the tasks. After this step, the receiving tasks can simultaneously add the values they received to the values they controlled, reducing the number of values to be added in half.

A single message-passing step is sufficient to combine two values into one. Two message-passing steps are sufficient to combine four values into two. In general, it is possible to perform a reduction of n values in $\log n$ message passing steps. See Figure 3.13, which illustrates **binomial trees** with one, two, four, and eight nodes. In a tree with $n = 2^k$ nodes, the maximum distance from any node to the root in the lower left corner is $k = \log n$. The binomial tree is one of the most common communication patterns in parallel algorithm design.

Figure 3.14 demonstrates how 16 tasks can combine their values in four communication steps when the channels are in the form of a binomial tree. In the first step, half of the tasks send values, and half of the tasks receive values. At this point the tasks that sent values become inactive, and the algorithm recurses on the remaining tasks. Half of the remaining tasks send values, and half of the remaining tasks receive values, and so on, until only a single task remains. This task, called the root, has the result of the reduction.

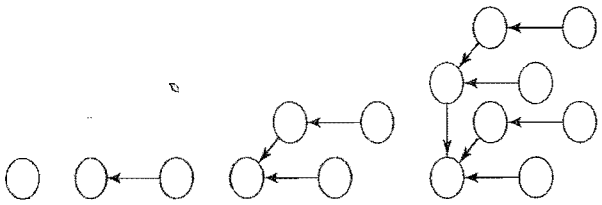


Figure 3.13 Binomial trees with 1, 2, 4, and 8 nodes.

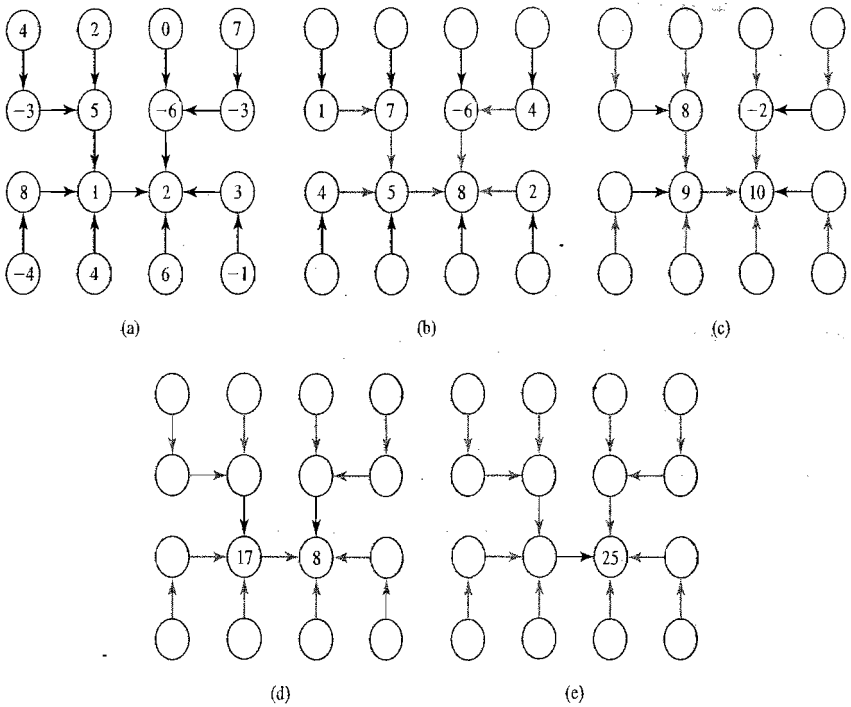


Figure 3.14 Finding the global sum in logarithmic time. (a) A task/channel graph forming a binomial tree. There is one task for each integer value in the list to be added. (b) Half of the tasks send values, and half of the tasks receive values and add. The sending tasks become inactive. (c) A quarter of the tasks send values, and a quarter of the tasks receive values and add. The sending tasks become inactive. (d) The process recurses with two sending tasks and two receiving/adding tasks. (e) In the final step, one task sends and one task receives and adds. The receiving/adding task has the grand total.

What if the number of tasks is not a power of 2? In this case, we modify the first step of the algorithm. Suppose the number of tasks $n = 2^k + r$ where $r < 2^k$. In the first step, r tasks send values, and r tasks receive values at which point r tasks become inactive. Once this step has been completed, the number of tasks with values is 2^k, and the previously described algorithm will work.

For example, consider a reduction among six tasks, as illustrated in Figure 3.15. In the first step two tasks send values to two other tasks. After this step four tasks have values, and the reduction can be done in $\log 4 = 2$ steps.

We see, then, that if the number of tasks n is a power of 2, reduction can be performed in $\log n$ communication steps. If n is not a power of 2, $\lfloor \log n \rfloor + 1$ communication steps are required. Hence in general the number of communication steps required for n tasks to perform a reduction is $\lceil \log n \rceil$.

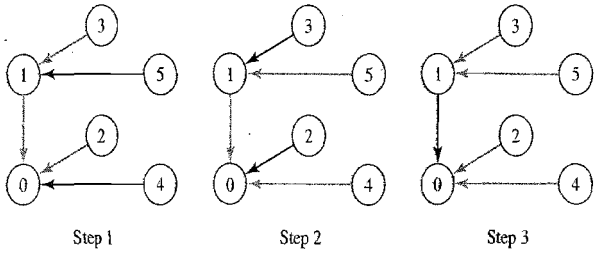


Figure 3.15 Example of reduction when the number of tasks is not a power of 2.

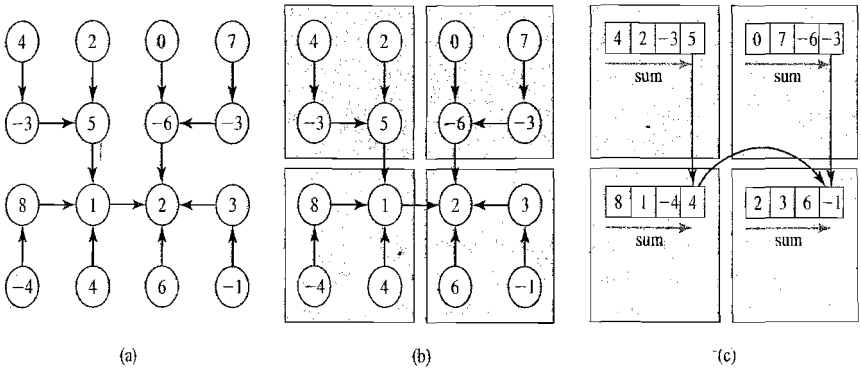


Figure 3.16 Example of agglomeration. (a) The original task/channel graph for the parallel reduction algorithm. (b) Sixteen tasks are mapped to four processors. Each processor has an equal number of tasks, and interprocessor communication is minimized. (c) The four tasks on each processor are agglomerated into a single task. Each task uses the sequential algorithm to find the local subtotal before communicating with the other tasks.

3.5.4 Agglomeration and Mapping

Figure 3.16a repeats the task/channel graph for a parallel reduction algorithm. Before implementing the algorithm as a parallel program, we need to perform a mapping of this n-task graph onto a set of p processors. To simplify our discussion, let's assume p is also a power of 2, but p is much less than n.

The number of tasks is static, computations per task are trivial, and the communication pattern is regular. Using the mapping decision tree of Figure 3.7, we conclude that we should agglomerate tasks to minimize communication. We can do this by assigning n/p "leaf" tasks to each of the p processors, as shown in Figure 3.16b.

As we agglomerate primitive tasks, there is no value to maintaining the illusion of separate tasks communicating with each other within a single physical

processor. The goal of this portion of the computation is simply to determine the sum of n/p values. Instead of n/p primitive tasks, each with a single value, we have a single task with n/p values. The result is shown in Figure 3.16c. The nice thing about this agglomeration is that it matches the one we have already chosen for the boundary value problem in the previous section. That means we can easily add this enhancement to our original parallel algorithm.

3.5.5 Analysis

At this point we can derive an expression for the expected running time of a parallel program to perform reduction. Let's define the following constants:

χ: time needed to perform the binary operation

λ: time needed to communicate an integer value from one task to another via a channel

If the n integers are divided evenly among the p tasks, no task will be responsible for more than $\lceil n/p \rceil$ integers. Since all tasks perform concurrently, the time needed for all the tasks to compute their subtotals is

$$(\lceil n/p \rceil - 1)\chi$$

We have already seen that a reduction of p values distributed among p tasks can be performed in $\lceil \log p \rceil$ communication steps. The receiving processor must not only wait for the message to arrive, it must also add the value it received to the value it already has. Hence each reduction step requires time

$$\lambda + \chi$$

Since there are $\lceil \log p \rceil$ communication steps, the overall execution time of the parallel program is

$$(\lceil n/p \rceil - 1)\chi + \lceil \log p \rceil (\lambda + \chi)$$

3.6 THE *n*-BODY PROBLEM

3.6.1 Introduction

Some problems arising in physics can be solved by performing computations on all pairs of objects in a dataset. For example, in some molecular dynamics problems the forces on the molecules may have a Coulombic or other long-range component. In a Newtonian n-body simulation, gravitational forces have infinite range. Straightforward sequential algorithms to solve these problems typically have time complexity $\Theta(n^2)$ per iteration, where n is the number of objects. While algorithms with significantly better time complexity have been developed for n-body problems, our focus here is on parallel algorithm development. For

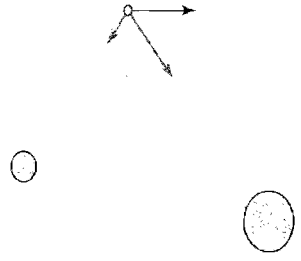


Figure 3.17 In the n-body problem every particle exerts a gravitational pull on every other particle. In this two-dimensional example, the white particle has a particular position and velocity vector (indicated by a black arrow). Its future position is influenced by the gravitational forces exerted by the other two particles.

this reason we consider the parallelization of a sequential algorithm in which a computation is performed on every pair of objects.

To ground our discussion, let's suppose we're solving an n-body problem. We are simulating the motion of n particles of varying mass in two dimensions. During each iteration of the algorithm we need to compute the new position and velocity vector of each particle, given the positions of all the other particles (Figure 3.17).

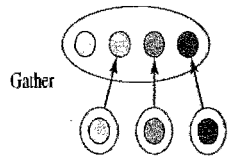
3.6.2 Partitioning

Our first step is to partition the dataset. To start with, let's assume we have one task per particle. In order for this task to compute the new location of the particle, it must know the locations of all the other particles.

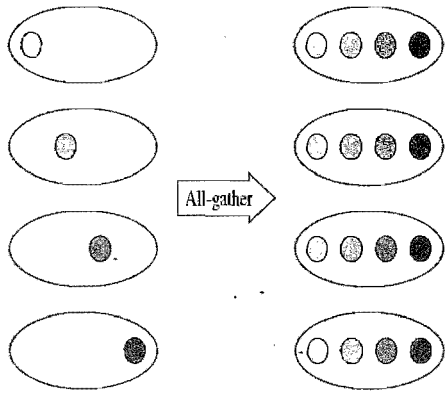
3.6.3 Communication

A **gather** operation is a global communication that takes a dataset distributed among a group of tasks and collects the items on a single task (Figure 3.18a). Unlike reduction, which computes a single result from the data elements, a gather operation results in the concatenation of the data items. An **all-gather** operation is similar to gather, except at the end of the communication every task has a copy of the entire dataset (Figure 3.18b).

In this case we want to update the location of every particle, so an all-gather communication is called for. One way to do this is to put a channel between every pair of tasks (Figure 3.19). During each communication step each task sends its vector element to one other task. After $n - 1$ communication steps, each task has



(a)



(b)

Figure 3.18 (a) The gather communication builds the concatenation of a set of data items on a single task. (b) The all-gather communication builds the concatenation of a set of data items on all tasks.

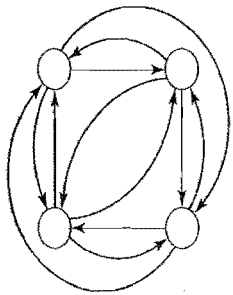


Figure 3.19 One way to make all data values available to all tasks is to set up a channel between every pair of tasks.

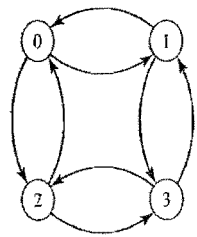


Figure 3.20 The all-gather data communication requires that each task have only log p outgoing channels and log p incoming channels.

the positions of all the other particles, and it can perform the calculations needed to determine the new location and velocity vector for its particle.

Is there a quicker way to get all values to all tasks? Inspired by parallel reduction, we ought to be looking for a way to perform the data routing in a logarithmic number of communication steps.

We can usually think about these algorithms from the top down or from the bottom up. We took a top-down approach to derive the reduction algorithm. For variety, let's try a bottom-up approach here.

Suppose there were only two particles. If each task has a single particle, they can exchange copies of their values. Each task sends one value on one channel and receives a value on another channel. What if there are four particles? After a single exchange step tasks 0 and 1 could both have particles v_0 and v_1, and tasks 2 and 3 could have particles v_2 and v_3. If task 0 now exchanges its pair of particles with task 2, while task 1 exchanges its pair of particles with task 3, all tasks will have all four particles. A task/channel graph for this improved algorithm appears in Figure 3.20.

A logarithmic number of exchange steps are necessary and sufficient to allow every processor to acquire the value originally held by every other processor. In the first exchange step the messages have length 1. In the second exchange step the messages have length 2. In the ith exchange step the messages have length 2^{i-1}.

The task/channel graph shown in Figure 3.20 is an example of a hypercube network, which we first encountered in Chapter 2. Task/channel graphs in the form of hypercubes often occur in efficient algorithms implementing various all-to-all data exchanges.

3.6.4 Agglomeration and Mapping

In general, there are far more particles n than processors p. Let's assume that n is a multiple of p. We associate one task per processor and agglomerate n/p particles into each task. Now the all-gather communication operation requires $\log p$ communication steps. In the first step the messages have length n/p, in the second step the messages have length $2n/p$, etc.

3.6.5 Analysis

Now we can derive an expression for the expected execution time of this algorithm. In the previous examples we assumed that it took λ units of time to send a message. However, in these examples the messages always had length 1. Now the messages can be much longer. It is unrealistic to expect that the time needed to send or receive a message is independent of the message length, so we'll add a new term to our formula for message-passing time. From now on λ (latency) will represent the time needed to initiate a message. Let β (bandwidth) represent the number of data items that can be sent down a channel in one unit of time. Sending a message containing n data items requires time $\lambda + n/\beta$ (Figure 3.21). Note that as bandwidth increases, communication time decreases.

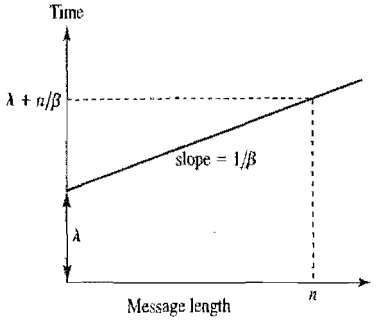


Figure 3.21 The time needed to send a message of length n is modeled by the linear function $\lambda + n/\beta$. The y intercept, λ, is the message latency, while β, the inverse of the slope, is the bandwidth of the communication system.

The communication time of the algorithm each iteration is

$$\sum_{i=1}^{\log p} \left(\lambda + \frac{2^{i-1}n}{\beta p}\right) = \lambda \log p + \frac{n(p-1)}{\beta p}$$

Each task is responsible for performing the gravitational force computation for n/p list elements. Suppose the time needed for this computation is χ. The computation time associated with the parallel algorithm each iteration is $\chi(n/p)$.

Putting together the communication time of the parallel convolution algorithm with the computation time, we derive an expected parallel execution time per iteration of

$$\lambda \log p + n(p-1)/(\beta p) + \chi(n/p)$$

3.7 ADDING DATA INPUT

3.7.1 Introduction

Most programs input and output data, yet the task/channel model as defined does not address data input/output. Let's consider how to add input and output to the n-body algorithm we have just developed. As we do so, we'll also add I/O channels to the basic model.

Let's suppose our parallel program will input the original positions and velocity vectors for the n particles. Commercial parallel computers often have parallel I/O systems, but commodity clusters often rely upon external file servers storing ordinary Unix files. For this reason, we set aside any notions of parallel I/O for

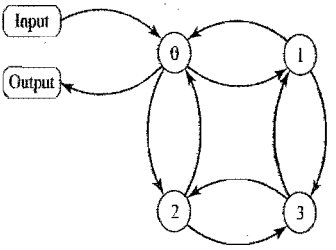


Figure 3.22 To show that task 0 is responsible for I/O, we augment the task/channel graph. I/O devices appear as rounded rectangles.

the time being and assume a single task is responsible for performing file I/O operations.

We augment the task/channel graph for the *n*-body problem, shown in Figure 3.20, by adding new channels for file I/O. The resulting task/channel graph appears in Figure 3.22. We'll give the task performing file I/O the rather obvious name of the I/O task. Note that we are not adding a new task to perform I/O. Instead, we are assigning additional duties to task 0.

The I/O task begins by opening the data file and reading the positions and velocities of the n particles. Since we're doing a two-dimensional simulation, a pair of coordinates identifies a particle's location, while its velocity can be represented by another pair of values. If $\lambda_{io} + n/\beta_{io}$ models the time needed to input or output n data elements, then reading the positions and velocities of all n particles requires time $\lambda_{io} + 4n\beta_{io}$.

3.7.2 Communication

After the I/O task inputs the particles, we must figure out how to break up the input data into pieces so that each task has its assigned subsection containing n/p elements. This global communication operation is called **scatter**. Can you see how a scatter operation is like a gather operation in reverse?

One way to scatter the particles is for the I/O task to simply send the correct n/p particles to each of the other tasks in turn. In other words, it sends $p - 1$ messages, each of length $4n/p$. The time required for this is

$$(p-1)\left(\lambda + 4n/(p\beta)\right)$$

This is not an efficient algorithm, because the communication is not balanced among the processors.

Using a process similar to what we have already done several times in this chapter, we can derive a scatter operation requiring $\log p$ communication steps. In the first step the I/O task sends half the list to another task. In the second step each task with a half list sends a quarter list to previously inactive tasks. Now

four tasks each have a quarter of the list. In step 3 the four tasks with quarter lists send eighth lists to four previously inactive tasks, and so on. The time required for this is

$$\sum_{i=1}^{\log p} \left(\lambda + 4n/(2^i p\beta)\right) = \lambda \log p + 4n(p-1)/(\beta p)$$

Now we have seen two different designs for the scatter algorithm. In the first algorithm one task sequentially sends $p-1$ messages to the other tasks. It requires time $(p-1)\lambda + 4n(p-1)/(\beta p)$.

The second algorithm works through about $\log p$ steps. (We fudge because p might not be an integer power of 2.) The total communication time is $\log(p)\lambda + 4n(p-1)/(\beta p)$. It is superior to the first algorithm.

Note that data transmission time (the term with the β) is identical for both algorithms. In the first algorithm each particle is passed directly to the task responsible for it. In the second algorithm particles are moved repeatedly. The typical particle is passed in about $\log p$ messages. Why, then, will a program based on the second algorithm spend no more time transmitting data than a program based on the first algorithm? *Our task/channel model supports the concurrent transmission of messages from multiple tasks, as long as they use different channels, and no two active channels have the same source or destination task.* This is a reasonable assumption on a commercial system. It is also a reasonable assumption on clusters in which each processor has a direct connection to a switch with sufficient backplane speed to support many concurrent messages between pairs of processors. It is not a reasonable assumption on a network of workstations connected by a hub or any shared communication medium that supports only a single message at a time.

3.7.3 Analysis

We can now derive an expression for the total expected execution time of the parallel n-body algorithm. The input and output of the positions and velocities of the n particles is a completely sequential operation requiring time

$$2(\lambda_{io} + 4n/\beta_{io})$$

Scattering the particles at the beginning of the algorithm and gathering the particles at the end of the computation require time

$$2\left(\lambda \log p + 4n(p-1)/(\beta p)\right)$$

Each iteration of the parallel algorithm requires an all-gather communication of the particles' positions. An implementation of this algorithm has approximate execution time

$$\lambda \log p + 2n(p-1)/(\beta p)$$

Finally, each processor performs its share of the computations. The expected execution time per iteration is

$$\chi\lceil(n/p)\rceil(n-1)$$

Suppose the algorithm executes for m iterations. The expected overall execution time of the parallel computation is about

$$2(\lambda_{io}+4n/\beta_{io})+2\big(\lambda\log p+4n(p-1)/(\beta p)\big)$$
$$+m\big(\lambda\log p+2n(p-1)/(\beta p)+\chi\lceil(n/p)\rceil(n-1)\big)$$

3.8 SUMMARY

The task/channel model described in this chapter is a theoretical construct that represents a parallel computation as a set of tasks that may interact with each other or I/O devices by sending messages through channels. This model is useful because it encourages parallel algorithm designs that maximize local computations and minimize communications, and these designs are a better fit for distributed-memory parallel computers.

In the process of developing a parallel algorithm for the task/channel model, the algorithm designer typically partitions the computation, identifies communications among primitive tasks, agglomerates primitive tasks into larger tasks, and decides how to map tasks to processors. The goals of this process are to maximize processor utilization by distributing the computational steps among the processors while minimizing interprocessor communications. Since neither goal can be reached without seriously compromising the other, good designs must strike a balance between them.

Reduction is the application of an associate binary operator across a dataset. Parallel algorithms often require reductions such as finding the grand total of values distributed across all the tasks. We developed a logarithmic-time parallel algorithm to perform reduction operations. The task/channel graph for reduction is in the form of a binomial tree.

We also developed an efficient parallel algorithm to form an all-gather operation, which provides every task with the concatenation of values collected from the entire set of tasks. Our algorithm requires only a logarithmic number of communication steps. It relies upon a task/channel graph in the form of a hypercube.

Finally, we considered the problem of scattering data on a single task among a set of tasks, as well as the inverse problem of gathering data distributed among a set of tasks back onto one task. The binomial tree is a suitable task/channel graph for scatter and gather operations when communication time is dominated by message latency.

3.9 KEY TERMS

agglomeration
all-gather
asynchronous
binomial tree
blocked task
channel
domain decomposition
functional decomposition
gather
global communication
increasing locality
local communication
mapping
partitioning
primitive task
processor utilization
reduction
scatter
synchronous
task

3.10 BIBLIOGRAPHIC NOTES

Part I of Foster's book, *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*, is devoted to developing a parallel algorithm design methodology based on the task/channel model [31]. He provides a more detailed treatment of the four-step design process (partitioning, communication, agglomeration, and mapping) presented in this chapter. The book also contains a variety of case studies illustrating the methodology.

Carriero and Gelernter present a much different view of parallel algorithm design. Their book, *How to Write Parallel Programs: A First Course*, describes three parallel algorithm design paradigms: result parallelism, specialist parallelism, and agenda parallelism [15]. Seeing the field of parallel algorithms from another perspective is a good way to enhance your capacity for "thinking in parallel."

Valiant has proposed the bulk synchronous parallel (BSP) model as a way of "bridging" the gap between parallel software and hardware [107]. The BSP model is designed to provide parallel algorithm designers the same benefits that the von Neumann model brings to the designers of sequential algorithms. A BSP computation is a sequence of supersteps. Each superstep consists of a sequence of steps in which processors perform computations on local data, followed by a barrier synchronization, when nonlocal data exchanges among the processors take place. For more information on BSP, check out the Web site for BSP Worldwide: `www.bsp-worldwide.org`.

3.11 EXERCISES

3.1 Give an example of how increasing processor utilization increases interprocessor communication.

3.2 Calculate $\log n$, $\lfloor \log n \rfloor$, and $\lceil \log n \rceil$ for the following values of n:

a. 3
b. 13
c. 32
d. 123
e. 321

3.3 Draw binomial trees of the following sizes:

a. 16 nodes

b. 32 nodes

3.4 Draw hypercubes of the following sizes, labeling the nodes:

a. 16 nodes

b. 32 nodes

3.5 Given a four-dimensional hypercube, draw four different subgraphs that are 16-vertex binomial trees. All four trees should be rooted at the same hypercube node. In each of these graphs, show the unused hypercube edges.

3.6 Illustrate how to perform a reduction in $\lceil \log n \rceil$ communication steps for the following values of n: 7, 11, 21.

3.7 Using the communication pattern illustrated in Figure 3.15 as your guide, write a C program that describes the communications performed by a task participating in a reduction. Given the number of tasks n and a task's particular identification number i, where $0 \leq i < n$, the program should print a list of messages sent and/or received by task i. The message list should indicate the destination task of all sent messages and the source task of all received messages.

For example, for the case where $n = 6$ and $i = 1$, the output of the program should be

```
Message received from task 5
Message received from task 3
Message sent to task 0
```

3.8 Prove that performing an n-element reduction on the task/channel model has time complexity $\Omega(\log n)$.

3.9 Many parallel algorithms require a broadcast step in which one task communicates a value it holds to all of the other tasks.

a. Using the task/channel model described in this chapter, devise an efficient parallel algorithm implementing broadcast.

b. Prove that the algorithm you devised in part (a) has optimal time complexity.

3.10 The all-gather algorithm we have developed routes n values to each of p tasks, while the scatter algorithm we have developed routes only about n/p values to each of p tasks, yet both algorithms have time complexity $\Theta(n + \log p)$. Explain.

3.11 Design a parallel algorithm to perform an *all-to-all* exchange. There are p processes where p is a power of 2. The processes are manipulating vectors of length p. Let $X_{i,j}$ denote the jth element of a vector controlled by process i. Each process begins with vector A. Each process ends with vector B, where $B_{i,j} = A_{j,i}$, in other words, the vector B held by process

i is the concatenation of the ith elements of all the A vectors. What is the complexity of your parallel algorithm?

3.12 The bubblesort algorithm sorts an array of keys $a[0], a[1], \ldots, a[n-1]$ by repeatedly comparing adjacent keys. If $a[i] > a[i+1]$, it exchanges the keys. The process continues until $a[0] < a[1] < \cdots a[n-1]$. Using the task/channel model, design a parallel version of bubblesort. Draw two task/channel diagrams for the parallel bubblesort. The first diagram should show primitive tasks. The second diagram should show agglomerated tasks.

3.13 A binary image is stored as an $n \times n$ array of 0s and 1s. The 1s represent objects, while the 0s represent empty space between objects. The component labeling problem is to associate a unique positive integer with every object. At the end of the algorithm, every 1-pixel will have a positive integer label. A pair of 1-pixels have the same label if and only if they are in the same component (object). The 1-pixels are in the same component if they are linked by a path of 1-pixels. Two 1-pixels are contiguous if they are adjacent to each other, either horizontally or vertically. Using the task/channel model, design a parallel algorithm solving the component labeling problem. Draw two task/channel diagrams for the parallel algorithm. The first diagram should show primitive tasks. The second diagram should show agglomerated tasks.

3.14 Given a crossword puzzle and a dictionary, design a parallel algorithm to find all possible ways to fill in the crossword puzzle so that every horizontal and vertical word space contains a word from the dictionary.

3.15 You are given an array of n records, each containing the x and y coordinates of a house. You are also given the x and y coordinates of a railroad station. Design a parallel algorithm to find the house closest to the railroad station (as the crow flies). Draw two task/channel diagrams. The first should show primitive tasks. The second should show agglomerated tasks.

3.16 The string matching problem is to find all occurrences of a particular substring, called the pattern, in another string, called the text. Design a parallel algorithm to solve the string matching problem.

3.17 Reconsider the string matching problem presented in the previous example. Suppose you were only interested in finding the first occurrence of the pattern in the text. How would that change the design of your parallel algorithm?

3.18 Given a list of n keys, $a[0], a[1], \ldots, a[n-1]$, all with distinct values, design a parallel algorithm to find the second-largest key on the list.

3.19 Given a list of n keys, $a[0], a[1], \ldots, a[n-1]$, design a parallel algorithm to find the second-largest key on the list. Note: Keys do not necessarily have distinct values.

CHAPTER

Message-Passing Programming

The voice of Nature loudly cries
And many a message from the skies,
That something in us never dies.
Robert Burns, *New Year's Day*

4.1 INTRODUCTION

Dozens of parallel programming languages have been introduced in the past 40 years. Many of them are high-level languages that simplify various aspects of managing parallelism. However, no single high-level parallel language has gained widespread acceptance in the parallel programming community. Instead, most parallel programming continues to be done in either Fortran or C augmented with functions that perform message-passing between processes. The MPI (Message Passing Interface) standard is the most popular message-passing specification supporting parallel programming. Virtually every commercial parallel computer supports MPI, and free libraries meeting the MPI standard are available for "homemade" commodity clusters.

In this chapter we begin a multiple-chapter introduction to parallel programming in C with MPI. Using the circuit satisfiability problem as an example, we design, write, enhance, and benchmark a simple parallel program. In doing so we introduce the following functions:

- `MPI_Init`, to initialize MPI
- `MPI_Comm_rank`, to determine a process's ID number
- `MPI_Comm_size`, to find the number of processes
- `MPI_Reduce`, to perform a reduction operation
- `MPI_Finalize`, to shut down MPI

- `MPI_Barrier`, to perform a barrier synchronization
- `MPI_Wtime`, to determine the time
- `MPI_Wtick`, to find the accuracy of the timer

4.2 THE MESSAGE-PASSING MODEL

The message-passing programming model is similar to the task/channel model we described in Chapter 3. See Figure 4.1. The underlying hardware is assumed to be a collection of processors, each with its own local memory. A processor has direct access only to the instructions and data stored in its local memory. However, an interconnection network supports message passing between processors. Processor A may send a message containing some of its local data values to processor B, giving processor B indirect access to these values.

A task in the task/channel model becomes a process in the message-passing model. The existence of the interconnection network means there is an implicit channel between every pair of processes; that is, every process can communicate with every other process. However, we will want to take advantage of the design

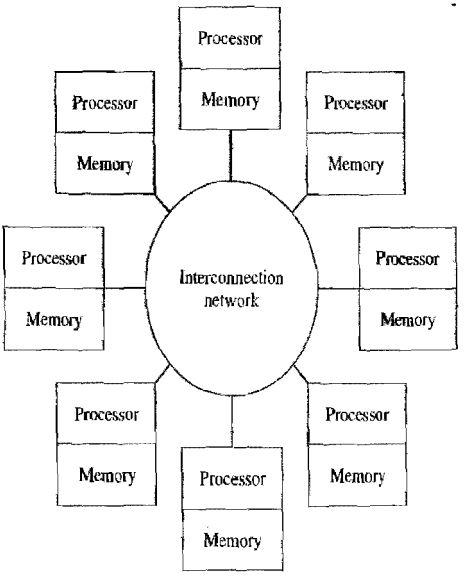


Figure 4.1 The message-passing model assumes that the underlying hardware is a collection of processors, each with its own local memory, and an interconnection network supporting message-passing between processors.

strategies we have learned in order to keep communication overhead from ruining the efficiency of our parallel programs.

The user specifies the number of concurrent processes when the program begins, and typically the number of active processes remains constant throughout the execution of the program. Every process executes the same program, but because each one has a unique ID number, different processes may perform different operations as the program unfolds. A process alternately performs computations on its local variables and communicates with other processes or I/O devices.

It is important to realize that in a message-passing model, processes pass messages both to communicate and to synchronize with each other. When a message containing data passes from one process to another, it obviously serves a communication function. A message has a synchronization function, too. Process B cannot receive a message from process A until after process A sends it. Hence receiving the message tells process B something about the state of process A. For this reason, even a message with no content has meaning.

Advocates of the message-passing model of parallel computation point toward several advantages it has over other parallel programming models. First, message-passing programs run well on a wide variety of MIMD architectures. They are a natural fit for multicomputers, which do not support a global address space. However, it is also possible to execute message-passing programs on multiprocessors by using shared variables as message buffers. In fact, the message-passing model's distinction between faster, directly accessible local memory and slower, indirectly accessible remote memory encourages designers to develop algorithms that maximize local computations while minimizing communications. The resulting programs tend to exhibit high cache hit rates when executing on multiprocessors, leading to good performance. Put another way, the message-passing model provides the multiprocessor programmer with the tools needed to manage the memory hierarchy.

Second, debugging message-passing programs is simpler than debugging shared-variable programs. Since each process controls its own memory, it is not possible for one process to accidentally overwrite a variable controlled by another process, a common bug in shared-variable programs. Nondeterministic execution (e.g., different processes accessing the same resource in different orders on several program executions) complicates debugging. In the message-passing model it is easier to construct a program that executes deterministically.

4.3 THE MESSAGE-PASSING INTERFACE

In the late 1980s many companies began manufacturing and selling multicomputers. Typically, the programming environment for one of these systems consisted of an ordinary sequential language (usually C or FORTRAN), augmented with a message-passing library enabling processes to communicate with each other. Each vendor had its own set of function calls, which meant that a program developed for an Intel iPSC, for example, could not be compiled and executed on an

nCUBE/10. Programmers did not appreciate this lack of portability, and after a few years there was a great deal of support for the creation of a message-passing library standard for parallel computers.

In the summer of 1989 the first version of a message-passing library called PVM (Parallel Virtual Machine) was written at Oak Ridge National Laboratory. PVM facilitated the execution of parallel programs across heterogeneous collections of serial and parallel computers. While the original version was used within Oak Ridge National Laboratory, it was not released to the public. Team members rewrote the software twice and released version 3 of PVM to the public in March 1993 [39]. PVM immediately became popular among parallel programmers.

Meanwhile, the Center for Research on Parallel Computing sponsored the Workshop on Standards for Message Passing in a Distributed Memory Environment in April 1992. This workshop attracted about 60 people from 40 organizations, primarily from the United States and Europe. Most major multicomputer vendors, along with researchers from universities, government laboratories, and industry, were represented. The group discussed basic features of a standard message-passing interface and created a working group to continue the standardization process. In November 1992 the preliminary draft proposal was completed. The Message Passing Interface Forum met from November 1992 to April 1994 to debate and refine the draft standard. Rather than simply adopt as a standard one of the many existing message-passing libraries, such as PVM or one of the commercial vendors' libraries, the MPI Forum attempted to pick and choose their best features. Version 1.0 of the standard, commonly referred to as MPI, appeared in May 1994. Since then, work has continued to evolve the standard, in particular to add parallel I/O and bindings to Fortran 90 and C++. MPI-2 was adopted in April 1997.

Today, MPI has become the most popular message-passing library standard for parallel programming. It is available on most commercial multicomputers. For those who are constructing their own multicomputers with commodity, off-the-shelf parts, free versions of MPI libraries are readily available over the Web from Argonne National Laboratory and other sites.

Writing parallel programs using MPI allows you to port them to different parallel computers, though the performance of a particular program may vary widely from one machine to another.

4.4 CIRCUIT SATISFIABILITY

For our initiation to MPI, we will implement a program that computes whether the circuit shown in Figure 4.2 is satisfiable. In other words, for what combinations of input values (if any) will the circuit output the value 1? The circuit-satisfiability problem is important for the design and verification of logical devices. Unfortunately, it is in the class NP-complete, which means there is no known polynomial time algorithm to solve general instances of this problem [38].

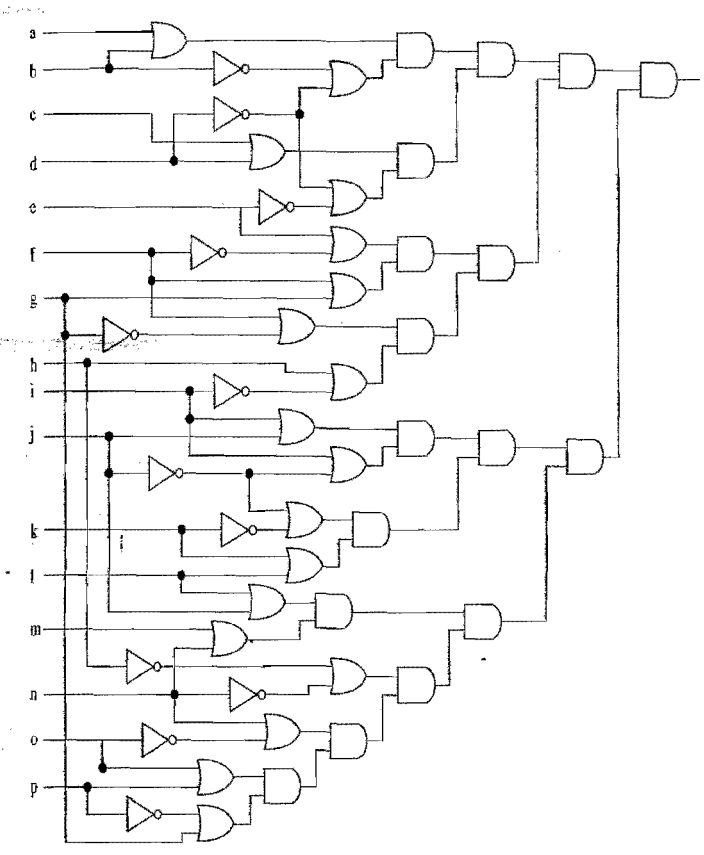


Figure 4.2 A circuit containing AND, OR, and NOT gates. The circuit satisfiability problem is to determine if some combination of inputs causes the output of the circuit to be 1.

One way to solve the problem is to try every combination of inputs. Since this circuit has 16 inputs, labeled a–p, and every input can take on two values, 0 and 1, there are $2^{16} = 65{,}536$ combinations of inputs.

As we saw in Chapter 3, the first step in parallel algorithm development is partitioning. Where is the parallelism? In this case, the parallelism is easy to spot. We need to test each of the 65,536 combinations of inputs on the circuit, to see if any of them result in the output value 1. A functional decomposition is natural for this application. We associate one task with each combination of inputs. If a task finds that its combination of inputs causes the circuit to return the value 1, it prints its combination. Since all of these tasks are independent, the satisfiability checks may be performed in parallel.

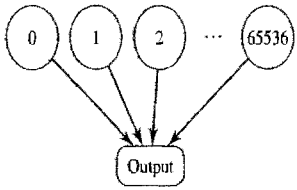


Figure 4.3 The task/channel graph for the circuit satisfiability problem. This is an example of an embarrassingly parallel problem, because there are no interactions among the tasks.

The task/channel graph for the circuit satisfiability problem appears in Figure 4.3. Since the tasks do not need to interact with each other, there are no channels between tasks. (Some people call this an **embarrassingly parallel** problem.) However, since any of the tasks may produce output, there is a channel from each task to the output device.

Our next step is to think about agglomeration and mapping. We have a fixed number of tasks. There are no communications between tasks. The time needed for each task to complete is variable. Can you see why? Nearly all tasks represent bit combinations for which the circuit is not satisfiable. With some bit patterns, we may quickly discover the circuit is not satisfiable. With others, it may take longer. Using the decision tree of Figure 3.7, we see that a good strategy is to map tasks to processors in a cyclic fashion in an effort to balance the computational load. Let's see how that strategy plays out for this program.

To minimize process creation time, we want to create one process per processor. So we have n pieces of work we want to allocate to p processes. A **cyclic (or interleaved) allocation** assigns each process every pth piece of work in a round-robin fashion. For example, suppose $n = 20$ and $p = 6$. Then process 0 would be responsible for indices 0, 6, 12, and 18; process 1 would be responsible for indices 1, 7, 13, and 19; process 2 would be responsible for indices 2, 8, and 14; process 3 would be responsible for indices 3, 9, and 15; process 4 would be responsible for indices 4, 10, and 16; and process 5 would be responsible for indices 5, 11, and 17.

Formally, if n pieces of work, labeled $0, 1, \ldots, n-1$ are to be assigned in a cyclic manner to p processes, labeled $0, 1, \ldots, p-1$, then work unit k is assigned to process k modulo p.

Before launching into C code, let's summarize the design of the program. We are going to determine whether the circuit shown in Figure 4.2 is satisfiable by considering all 65,536 combinations of the 16 boolean inputs. The combinations will be allocated in a cyclic fashion to the p processes. Every process will examine each of its combinations in turn. If a process finds a combination of inputs that satisfies the circuit, it will print that combination of inputs.

Now let's take a close look at the C code. (The entire program appears in Figure 4.4.)

The program begins with preprocessor directives to include the header files for MPI and standard I/O.

```
#include <mpi.h>
#include <stdio.h>
```

Next comes the header for function `main`. Note that we include the `argc` and `argv` parameters, which we will need to pass to the function that initializes MPI.

```
int main (int argc, char *argv[]) {
```

Function `main` has three scalar variables. Variable `i` is the loop index, `id` is the process ID number, and `p` is the number of active processes. Remember if there are p processes, then the ID numbers start at 0 and end at $p - 1$.

Each active MPI process executes its own copy of this program. That means each MPI process has its own copy of all of the variables declared in the program, whether they be external variables (declared outside of any function) or automatic variables declared inside a function.

We also include the prototype for function `check_circuit`, which will determine if the ith combination of inputs satisfies the circuit.

4.4.1 Function `MPI_Init`

The first MPI function call made by every MPI process is the call to `MPI_Init`, which allows the system to do any setup needed to handle further calls to the MPI library. The call to `MPI_Init` does not have to be the first executable statement of the program. In fact, it does not even have to be located in function `main`. The only requirement is that `MPI_Init` be called before any other MPI function.[1]

Note that all MPI identifiers, including function identifiers, begin with the prefix `MPI_`, followed by a capital letter and a series of lowercase letters and underscores. All MPI constants are strings of capital letters and underscores beginning with `MPI_`.

```
MPI_Init (&argc, &argv);
```

4.4.2 Functions `MPI_Comm_rank` and `MPI_Comm_size`

When MPI has been initialized, every active process becomes a member of a communicator called `MPI_COMM_WORLD`. A **communicator** is an opaque object that provides the environment for message passing among processes. `MPI_COMM_WORLD` is the default communicator that you get "for free." For

[1] The exception to this statement is that function `MPI_Initialized`, which checks to see if MPI has been initialized, may be called before `MPI_Init`.

```
/*
 * Circuit Satisfiability, Version 1
 *
 *   This MPI program determines whether a circuit is
 *   satisfiable, that is, whether there is a combination of
 *   inputs that causes the output of the circuit to be 1.
 *   The particular circuit being tested is "wired" into the
 *   logic of function 'check_circuit'. All combinations of
 *   inputs that satisfy the circuit are printed.
 *
 *   Programmed by Michael J. Quinn
 *
 *   Last modification: 3 September 2002
 */

#include <mpi.h>
#include <stdio.h>

int main (int argc, char *argv[]) {
   int i;
   int id;                 /* Process rank */
   int p;                  /* Number of processes */
   void check_circuit (int, int);

   MPI_Init (&argc, &argv);
   MPI_Comm_rank (MPI_COMM_WORLD, &id);
   MPI_Comm_size (MPI_COMM_WORLD, &p);

   for (i = id; i < 65536; i += p)
      check_circuit (id, i);

   printf ("Process %d is done\n", id);
   fflush (stdout);
   MPI_Finalize ();
   return 0;
}
```

Figure 4.4 Version 1 of MPI program to solve the circuit satisfiability problem.

most of our programs, it is sufficient. However, you can create your own communicators if you need to partition the processes into independent communication groups. You'll see how to do this in Chapter 8.

Processes within a communicator are ordered. The **rank** of a process is its position in the overall order. In a communicator with p processes, each process has a unique rank (ID number) between 0 and $p - 1$. A process may use its rank to determine which portion of a computation and/or a dataset it is responsible for.

A process calls function `MPI_Comm_rank` to determine its rank within a communicator. It calls `MPI_Comm_size` to determine the total number of processes in a communicator.

```
MPI_Comm_rank (MPI_COMM_WORLD, &id);
MPI_Comm_size (MPI_COMM_WORLD, &p);
```

```
/* Return 1 if 'i'th bit of 'n' is 1; 0 otherwise */
#define EXTRACT_BIT(n,i) ((n&(1<<i))?1:0)

void check_circuit (int id, int z) {
   int v[16];        /* Each element is a bit of 'z' */
   int i;

   for (i = 0; i < 16; i++) v[i] = EXTRACT_BIT(z,i);

   if ((v[0] || v[1]) && (!v[1] || !v[3]) && (v[2] || v[3])
      && (!v[3] || !v[4]) && (v[4] || !v[5])
      && (v[5] || !v[6]) && (v[5] || v[6])
      && (v[6] || !v[15]) && (v[7] || !v[8])
      && (!v[7] || !v[13]) && (v[8] || v[9])
      && (v[8] || !v[9]) && (!v[9] || !v[10])
      && (v[9] || v[11]) && (v[10] || v[11])
      && (v[12] || v[13]) && (v[13] || !v[14])
      && (v[14] || v[15])) {
      printf ("%d) %d%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d\n", id,
         v[0],v[1],v[2],v[3],v[4],v[5],v[6],v[7],v[8],v[9],
         v[10],v[11],v[12],v[13],v[14],v[15]);
      fflush (stdout);
   }
}
```

Figure 4.4 (contd.) Version 1 of MPI program to solve the circuit satisfiability problem.

Now that the MPI process knows its rank and the total number of processes, it may check its share of the 65,536 possible inputs to the circuit.

```
for (i = id; i < 65536; i += p)
   check_circuit (id, i);
```

After the process has completed the loop, it has no more work to do, and it prints a message indicating that it is done. We put a call to `fflush` after every `printf` statement. This flushes the output buffer and helps ensure the eventual appearance of the message on standard output, even if the parallel program crashes.

```
printf ("Process %d is done\n", id);
fflush (stdout);
```

4.4.3 Function `MPI_Finalize`

After a process has completed all of its MPI library calls, it calls function `MPI_Finalize`, allowing the system to free up resources (such as memory) that have been allocated to MPI.

```
   MPI_Finalize();
   return 0;
}
```

Function `check_circuit`, passed the ID number of a process and an integer `z`, first extracts the values of the 16 inputs using the macro `EXTRACT_BITS`. Element `v[0]` corresponds to input `a`, element `v[1]` corresponds to input `b`, and so on. Calling function `check_circuit` with values of `z` ranging from 0 through 65,535 generates all 2^{16} combinations of values.

After function `check_circuit` has determined the values of the 16 inputs, it checks to see if they result in the circuit having the output 1. If so, the process prints the values of `a` through `p`.

4.4.4 Compiling MPI Programs

After entering this program into file `sat1.c`, we need to compile it. The command to compile an MPI program varies from system to system. Here is a common command-line syntax:

```
% mpicc -o sat1 sat1.c
```

With this command the system compiles the MPI program stored in file `sat1.c` and stores the executable in file `sat1`.

4.4.5 Running MPI Programs

The typical command for running an MPI program is `mpirun`. The `-np` flag indicates the number of processes to create. Let's examine the output when we execute the program using a single process:

```
% mpirun -np 1 sat1
0) 1010111110011001
0) 0110111110011001
0) 1110111110011001
0) 1010111111011001
0) 0110111111011001
0) 1110111111011001
0) 1010111110111001
0) 0110111110111001
0) 1110111110111001
Process 0 is done
```

The program identifies nine combinations of inputs that will satisfy the circuit (i.e., cause it to have output value 1). For example, the first line of output indicates that the circuit is satisfied when `a`, `c`, `e`, `f`, `g`, `h`, `i`, `l`, `m`, and `p` are true (have the value 1) and the other variables are false (have the value 0). Note that the output of the parallel program on a single process is identical to the output of a sequential program solving the same problem, since the lone process evaluates the combinations in the same order as a sequential program.

Now let's look at the output of the program when we execute it using two processes:

```
% mpirun -np 2 sat1
0) 0110111110011001
0) 0110111111011001
0) 0110111110111001
1) 1010111110011001
1) 1110111110011001
1) 1010111111011001
1) 1110111111011001
1) 1010111110111001
1) 1110111110111001
Process 0 is done
Process 1 is done
```

Together, the two processes identified all nine solutions, but process 0 found three of them, while process 1 found six.

Here is the result of an execution with three processes:

```
% mpirun -np 3 sat1
0) 0110111110011001
0) 1110111111011001
2) 1010111110011001
1) 1110111110011001
1) 1010111111011001
1) 0110111110111001
0) 1010111110111001
2) 0110111111011001
2) 1110111110111001
Process 1 is done
Process 2 is done
Process 0 is done
```

Again, all nine solutions were found. It turns out that each process found three of the solutions. Note that output of the processes is mixed up in a haphazard fashion.

The order in which output appears on standard output only partially reflects the order in which the output events actually occurred inside the parallel computer. If process A prints two messages to standard output, then the first message will be printed before the second message. However, if process A prints to standard output before process B prints to standard output, that does not guarantee process A's output will appear before the output of process B.

Assuming that the order in which messages appear is the same as the order in which the `printf` statements executed can lead to false conclusions about the execution of the parallel program, making the search for bugs much more difficult. Avoid this mental trap!

4.5 INTRODUCING COLLECTIVE COMMUNICATION

We are off to a good start. We have our first MPI program up and running. However, it is not hard to find ways to improve it. For example, what if we want to know how many different ways a circuit can be satisfied? In our previous example, there were only nine solutions, and it was easy for us to count them by hand, but what if there had been 99?

For our next program, we want to add functionality that will enable the processes to compute the total number of solutions. It is easy enough for a single process to maintain an integer variable accumulating the number of solutions it has found, but then the processors must cooperate to compute the global sum of these values.

A **collective communication** is a communication operation in which a group of processes works together to distribute or gather together a set of one or more values. Reduction is an example of an operation that requires collective communication in a message-passing environment.

We will modify our first circuit satisfiability program to compute the total number of solutions to the circuit. The new version of function `main` appears in Figure 4.5.

Let's go through the changes we have made to function `main`. First, we introduce two new integer variables. Integer `solutions` keeps track of the number of solutions this process has found. Process 0 (and only processor 0) will use integer variable `global_solutions` to store the grand total of the count values of all the MPI processes. It will be responsible for printing the count at the end of the program's execution.

```
int solutions;
int global_solutions;
```

We must modify function `check_circuit` to return the value 1 if the particular combination satisfies the circuit. It should return the value 0 if the combination does not satisfy the circuit. This modification to the function is trivial, and we will not discuss it further.

```
int check_circuit (int, int);
```

We modify the `for` loop to accumulate the number of valid solutions this process discovers.

```
solutions = 0;
for (i = id; i < 65536; i += p)
   solutions += check_circuit (id, i);
```

```
/*
 *   Circuit Satisfiability, Version 2
 *
 *   This enhanced version of the program also prints the
 *   total number of solutions.
 */

#include "mpi.h"
#include <stdio.h>

int main (int argc, char *argv[]) {
   int global_solutions;     /* Total number of solutions */
   int i;
   int id;                   /* Process rank */
   int p;                    /* Number of processes */
   int solutions;            /* Solutions found by this proc */
   int check_circuit (int, int);

   MPI_Init (&argc, &argv);
   MPI_Comm_rank (MPI_COMM_WORLD, &id);
   MPI_Comm_size (MPI_COMM_WORLD, &p);

   solutions = 0;
   for (i = id; i < 65536; i += p)
      solutions += check_circuit (id, i);

   MPI_Reduce (&solutions, &global_solutions, 1, MPI_INT, MPI_SUM, 0,
      MPI_COMM_WORLD);
   printf ("Process %d is done\n", id);
   fflush (stdout);
   MPI_Finalize ();
   if (id == 0) printf ("There are %d different solutions\n",
      global_solutions);
   return 0;
}
```

Figure 4.5 Version 2 of MPI program to solve circuit satisfiability problem. In this version the processes collectively determine the number of solutions to the problem.

4.5.1 Function `MPI_Reduce`

After a process has completed its share of the work, it is ready to participate in the reduction operation. Function `MPI_Reduce` performs one or more reduction operations on values submitted by all the processes in a communicator. The header for function `MPI_Reduce` is

```
int MPI_Reduce (
   void         *operand, /* addr of 1st reduction element */
   void         *result,  /* addr of 1st reduction result */
   int          count,    /* reductions to perform */
   MPI_Datatype type,     /* type of elements */
   MPI_Op       operator, /* reduction operator */
   int          root,     /* process getting result(s) */
   MPI_Comm     comm)     /* communicator */
```

Let's consider each of the function parameters. The third parameter, `count`, indicates how many reductions are being performed. Each process submits `count` values, and each of these values is a list element for a different reduction.

Parameter `operand` is an input parameter. The calling process indicates the location of its element for the first reduction. If `count` is greater than 1, then the list elements for all of the reductions occupy a contiguous block of memory.

Parameter 4, `type`, is an input parameter designating the type of the elements being reduced. A list of the MPI constants and their associated C types appears in Table 4.1.

The fifth parameter, `operator`, indicates the kind of reduction to perform. A list of all built-in reduction operators appears in Table 4.2.

The sixth parameter, `root`, gives the rank of the process that will have the results of all the reductions.

Parameter `result` points to the location of the first reduction result. This parameter only has meaning for process `root`.

The last parameter, `comm`, gives the name of the communicator—that is, the set of processes participating in the reduction.

Table 4.1 MPI constants for C data types.

Name	C type
MPI_CHAR	signed char
MPI_DOUBLE	double
MPI_FLOAT	float
MPI_INT	int
MPI_LONG	long
MPI_LONG_DOUBLE	long double
MPI_SHORT	short
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long
MPI_UNSIGNED_SHORT	unsigned short

Table 4.2 MPI's built-in reduction operators.

Name	Meaning
MPI_BAND	Bitwise and
MPI_BOR	Bitwise or
MPI_BXOR	Bitwise exclusive or
MPI_LAND	Logical and
MPI_LOR	Logical or
MPI_LXOR	Logical exclusive or
MPI_MAX	Maximum
MPI_MAXLOC	Maximum and location of maximum
MPI_MIN	Minimum
MPI_MINLOC	Minimum and location of minimum
MPI_PROD	Product
MPI_SUM	Sum

Our particular call to MPI_Reduce takes the form:

```
MPI_Reduce (&solutions, &global_solutions, 1, MPI_INT, MPI_SUM, 0,
   MPI_COMM_WORLD);
```

After this function returns, process 0 has in its variable global_solutions the sum of all of the processes' values of variable solutions. It prints the global sum after the call to MPI_Finalize, so that it will appear at the end of the output of the program.

It is crucially important to remember that while only a single process (in this case process 0) gets the global result, every process must call function MPI_Reduce. There is no magic! Every process in the communicator must enter the reduction voluntarily—it cannot be "summoned" by process 0. If you write a program in which not all the processes in a communicator call MPI_Reduce or any other collective communication function, the program will "hang" at the point that function is executed, unable to complete it.

So now we're back from MPI_Reduce. Note that we conditionalize execution of the printf function call so that only process 0 prints the value of global_solutions. We do this for two reasons. First, only process 0 has the actual global sum in its variable global_solutions. The value of this variable for the other processes is undefined. Second, even if every process had the correct sum in its copy of global_solutions, how many times do you want to read the answer? It is sufficient for one process to print the solution.

```
if (id==0) printf ("There are %d different solutions\n",
   global_solutions);
```

Here is an example of the program executing on three processes:

```
% mpirun -np 3 sat2
0) 0110111110011001
0) 1110111111011001
1) 1110111110011001
1) 1010111111011001
2) 1010111110011001
2) 0110111111011001
2) 1110111110111001
1) 0110111110111001
0) 1010111110111001
Process 1 is done
Process 2 is done
Process 0 is done
There are 9 different solutions
```

Compare this output with the output of the first program executing on three processes. Although each process finds the same solutions in the same order, the order in which the processes' output appears is different.

4.6 BENCHMARKING PARALLEL PERFORMANCE

Now that we have a parallel program up and running, it is only natural to ask whether we are benefitting from parallel execution. In other words, are we getting the results any quicker?

4.6.1 Functions `MPI_Wtime` and `MPI_Wtick`

One way to measure the performance of a parallel application is to look at the wall clock time, measuring the number of seconds that elapse from the time we initiate execution until the program terminates. In production environments, this may be the most useful metric.

For our purposes, however, we would like to take a narrower focus. Typically, we are going to ignore the time spent initiating MPI processes, establishing communications sockets between them, and performing I/O on sequential devices. Instead, we will measure how well our parallel programs stack up against their sequential counterparts in the "middle area" between reading the dataset and writing the results.

MPI provides a function called `MPI_Wtime` that returns the number of seconds that have elapsed since some point of time in the past. Function `MPI_Wtick` returns the precision of the result returned by `MPI_Wtime`. Here are the headers of these two functions:

```
double MPI_Wtime (void)
double MPI_Wtick (void)
```

We can benchmark a section of code by putting a pair of calls to function `MPI_Wtime` before and after the section. The difference between the two values returned by the function is the number of seconds elapsed.

From a logical point of view, every MPI process begins execution at the same time, but this is not true in practice. MPI processes executing on different processors may begin executing seconds apart. This can throw off timings significantly. For example, in the case of our second program for circuit satisfiability, the processes call `MPI_Reduce` to find the total number of solutions. Since all processes must participate in this communication function, no process may complete the function until all processes have reached it. Processes that began execution early may wait around quite a while before the stragglers catch up. These processes will report significantly longer computation times than the latecomers.

4.6.2 Function `MPI_Barrier`

We address this problem by introducing a barrier synchronization before the first call to `MPI_Wtime`. Recall that no process can proceed beyond a barrier until all processes have reached it. Hence a barrier ensures that all processes are going into the measured section of code at more or less the same time.

Here is the prototype for the barrier function:

```
int MPI_Barrier (MPI_Comm comm)
```

The single argument to `MPI_Barrier` indicates the communicator participating in the barrier.

We can benchmark our circuit satisfiability program by adding a local variable to function `main`:

```
double elapsed_time;
```

We start the timer after initializing MPI:

```
MPI_Init (&argc, &argv);
MPI_Barrier (MPI_COMM_WORLD);
elapsed_time = - MPI_Wtime();
```

After the call to `MPI_Reduce` we stop the timer:

```
MPI_Reduce (&solutions, &global_solutions, 1, MPI_INT, MPI_SUM, 0,
   MPI_COMM_WORLD);
elapsed_time += MPI_Wtime();
```

Since we do not want to count I/O time, we also need to comment out the calls to `printf` and `fflush` inside function `check_circuit`.

Now we are ready to benchmark the program. The results appear as the solid line in Figure 4.6. As we add processors, execution time decreases, because each

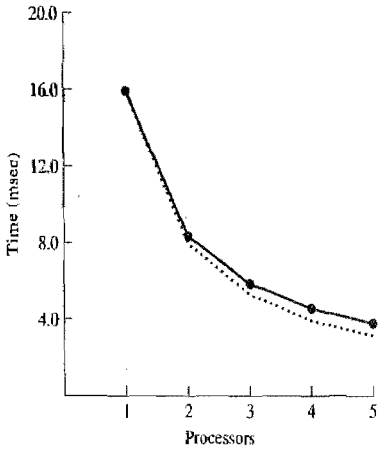


Figure 4.6 Mean execution time of second circuit satisfiability program on a commodity cluster with 450 MHz Pentium II CPUs connected by fast Ethernet. The dashed line indicates a "perfect" speed improvement, in which p processors execute the program p times as fast as one processor.

processor is responsible for checking a smaller number of circuits. The dashed line in the figure shows what the execution time would have been if two processors had executed the program in half the time, three processors had executed the program in one-third the time, etc. The reason that actual execution time is greater than this is because some time is spent performing the sum reduction at the end of the program. This communication time represents an overhead not incurred by the sequential program. As the number of processors grows, this overhead grows, too.

4.7 SUMMARY

Designing parallel algorithms using the task/channel model leads naturally to their implementation using the message-passing programming paradigm. The message-passing paradigm allows programmers to control memory utilization and increase locality. Since keeping most memory references local is a key strategy for optimizing performance on both multicomputers and multiprocessors, message-passing programs can run efficiently on a wide range of parallel systems.

In the early years of commercial multicomputers, every manufacturer had its own message-passing library, but programmers called for a standard to increase the portability of programs. The Message Passing Interface (MPI) standard is the result of a collaborative effort between companies and researchers. Today, nearly every commercial computer supports MPI functions. MPI libraries are also freely available to those constructing commodity clusters.

We have developed and benchmarked a parallel program to solve an instance of the circuit satisfiability problem. In the process, we have used a small set of MPI library functions. In future chapters we will add significantly to the number of MPI functions in our repertoire.

4.8 KEY TERMS

collective communication
communicator
cyclic (or interleaved) allocation
embarrassingly parallel
rank

4.9 BIBLIOGRAPHIC NOTES

Computer scientists have been contemplating parallel programming since the dawn of the computer age. In fact, the lead article in the inaugural issue of the British Computer Society's *Computer Journal* has the title "Parallel Programming" [40]. The issue is dated April 1958.

Contemporary introductions to programming using MPI include Pacheco's *Parallel Programming with MPI* [89] and Gropp et al.'s *Using MPI: Portable Parallel Programming with the Message-Passing Interface* [45]. In addition, Foster has a chapter on MPI in his book on parallel algorithm design [31].

In Chapter 5 of *Practical Parallel Programming*, Wilson summarizes the features and highlights the shortcomings of three different kinds of message-passing models [116]. In the first model anonymous processes are connected by channels. The programming language occam is based on this model. In the second model processes are organized into a regular topology and may communicate only with their neighbors or a control process. In the third model processes have names, and any process may communicate with any other. MPI falls into this category.

4.10 EXERCISES

4.1 Suppose n pieces of work are allocated in cyclic fashion to p processes.

a. Which pieces of work are assigned to process k, where $0 \leq k \leq p-1$?

b. Which process is responsible for piece of work j, where $0 \leq j \leq n-1$?

c. What are the most pieces of work assigned to any process?

d. Identify all processes having the most pieces of work.

e. What are the fewest pieces of work assigned to any process?

f. Identify all processes having the fewest pieces of work.

4.2 Given a set of five unsigned, eight-bit integers with decimal values 13, 22, 43, 64, and 99, determine the decimal result of the following reductions:

a. add

b. multiply

c. maximum

d. minimum

e. bitwise *or*

f. bitwise *and*

g. logical *or*

h. logical *and*

Assume the meaning of the *and* and *or* operators is the same as in the C programming language.

4.3 Modify function `check_circuit` so that it returns the integer 1 if the input argument represents a satisfiable circuit, and 0 if the input argument does not represent a satisfiable circuit.

4.4 a. Benchmark the second circuit satisfiability program on your parallel computer for $1, 2, \ldots, 8$ processors (with printing disabled). For each number of processors, determine the mean execution time after five runs.

b. Summarize and interpret the results you observed.

4.5 The circuit satisfiability program presented in this chapter has the circuit to be tested "hard wired" into function `check_circuit`. Explain how

the circuit satisfiability program could be modified to check the satisfiability of a circuit input from a data file.

a. How would you represent a circuit with *and*, *or*, and *not* gates in a plain text file that could be created and viewed with a text editor?

b. How would your program parse this file?

c. Describe the data structure you would use to represent the circuit.

4.6 Write a parallel variant of Kernighan and Ritchie's classic "hello, world" program [61]. Each process should print a message of the form

```
hello, world, from process <i>
```

where <i> is its rank.

4.7 Write a parallel program that computes the sum $1 + 2 + \cdots + p$ in the following manner: Each process i assigns the value $i + 1$ to an integer, and then the processes perform a sum reduction of these values. Process 0 should print the result of the reduction. As a way of double-checking the result, process 0 should also compute and print the value $p(p + 1)/2$.

4.8 A prime number is a positive integer evenly divisible by exactly two positive integers: itself and 1. The first five prime numbers are 2, 3, 5, 7, and 11. Sometimes two consecutive odd numbers are both prime. For example, the odd integers following 3, 5, and 11 are all prime numbers. However, the odd integer following 7 is not a prime number. Write a parallel program to determine, for all integers less than 1,000,000, the number of times that two consecutive odd integers are both prime.

4.9 The gap between consecutive prime numbers 2 and 3 is only 1, while the gap between consecutive primes 7 and 11 is 4. Write a parallel program to determine, for all integers less than 1,000,000, the largest gap between a pair of consecutive prime numbers.

4.10 A small college wishes to assign unique identification numbers to all of its present and future students. The administration is thinking of using a six-digit identifier, but is not sure that there will be enough combinations, given various constraints that have been placed on what is considered to be an "acceptable" identifier. Write a parallel program to count the number of different six-digit combinations of the numerals 0–9, given these constraints:

- The first digit may not be a 0.
- Two consecutive digits may not be the same.
- The sum of the digits may not be 7, 11, or 13.

4.11 The value of the definite integral

$$\int_0^1 \frac{4}{1+x^2}\,dx$$

is π. We can use numerical integration to compute π by approximating the area under the curve. A simple way to do this is called the **rectangle**

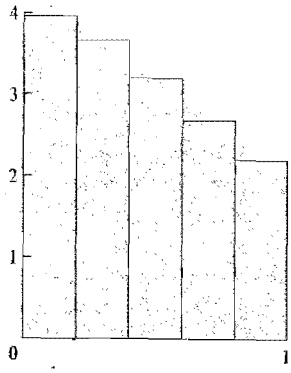


Figure 4.7 The rectangle rule is a simple way to approximate the area under a curve. In this example the function is $4/(1+x^2)$, and the area under the curve between 0 and 1 is π.

rule (Figure 4.7). We divide the interval [0, 1] into k subintervals of equal size. We find the height of the curve at the midpoint of each of these subintervals. With these heights we can construct k rectangles. The area of the rectangles approximates the area under the curve. As k increases, the accuracy of the estimate also increases.

A C program that uses the rectangle rule to approximate π appears in Figure 4.8.

a. Write a parallel program to compute π using the rectangle rule with 1,000,000 intervals.

b. Benchmark your program on various numbers of processors.

12 Simpson's Rule is a better numerical integration algorithm than the rectangle rule because it converges more quickly. Suppose we want to compute $\int_a^b f(x)\,dx$. We divide the interval $[a, b]$ into n subintervals, where n is even. Let x_i denote the end of the ith interval, for $1 \le i \le n$, and let x_0 denote the beginning of the first interval. According to Simpson's Rule:

$$\int_a^b f(x)\,dx \approx \frac{1}{3n}\left[f(x_0) - f(x_n) + \sum_{i=1}^{n/2}\left(4f(x_{2i-1}) + 2f(x_{2i})\right)\right]$$

A C program that uses Simpson's Rule to compute π appears in Figure 4.9.

a. Write a parallel program to compute the value of π using Simpson's Rule: $f(x) = 4/(1+x^2)$, $a = 0$, $b = 1$, and $n = 50$.

b. Benchmark your program on various numbers of processors.

```
/* This program computes pi using the rectangle rule. */

#define INTERVALS 1000000

int main (int argc, char *argv[])
{
   double area;  /* Area under curve */
   double ysum;  /* Sum of rectangle heights */
   double xi;    /* Midpoint of interval */
   int i;

   ysum = 0.0;
   for (i = 0; i < INTERVALS; i++) {
      xi = (1.0/INTERVALS)*(i+0.5);
      ysum += 4.0/(1.0+xi*xi);
   }
   area = ysum * (1.0 / INTERVALS);
   printf ("Area is %13.11f\n", area);
   return 0;
}
```

Figure 4.8 A C program to compute the value of π using the rectangle rule.

```
/* This program uses Simpson's Rule to compute pi. */

#define n 50

double f (int i) {
   double x;
   x = (double) i / (double) n;
   return 4.0 / (1.0 + x * x);
}

int main (int argc, char *argv[]) {
   double area;
   int i;
   area = f(0) - f(n);
   for (i = 1; i <= n/2; i++)
      area += 4.0*f(2*i-1) + 2*f(2*i);
   area /= (3.0 * n);
   printf ("Approximation of pi: %13.11f\n", area);
   return 0;
}
```

Figure 4.9 A C program to compute the value of π using Simpson's Rule.

CHAPTER

5

The Sieve of Eratosthenes

He was not merely a chip of the old block, but the old block itself.
Edmund Burke

5.1 INTRODUCTION

The Sieve of Eratosthenes is a useful vehicle for advancing to the next level of parallel programming with MPI. After an explanation of the sequential algorithm, we will use the domain decomposition methodology to come up with a data-parallel algorithm. During the task agglomeration step we will weigh the pros and cons of several schemes to allocate contiguous blocks of array elements to tasks. The resulting algorithm requires a broadcast step, and we will learn the syntax of an MPI function to perform the broadcast.

After coding and benchmarking an initial parallel program, we will consider three ways to improve its performance, including using redundant computations to reduce process communication time and rearranging the order of computations to increase the cache hit rate. Benchmarking these program improvements highlights the importance of maximizing single-processor performance, even when multiple processors are available.

This chapter introduces the following MPI function:

`MPI_Bcast`, to broadcast a message to all processes in a communicator

5.2 SEQUENTIAL ALGORITHM

Our goal is to develop a parallel version of the prime sieve invented by the Greek mathematician Eratosthenes (276-194 BCE). You can find pseudocode for the Sieve of Eratosthenes in Figure 5.1.

An example of the sieve appears in Figure 5.2. In order to find primes up to 60, integer multiples of the primes 2, 3, 5, and 7 are marked as composite

1. Create a list of natural numbers 2, 3, 4, . . . , n, none of which is marked.
2. Set k to 2, the first unmarked number on the list.
3. Repeat
   (a) Mark all multiples of k between k^2 and n
   (b) Find the smallest number greater than k that is unmarked. Set k to this new value.
   Until $k^2 > n$
4. The unmarked numbers are primes.

Figure 5.1 The Sieve of Eratosthenes finds primes between 2 and n.

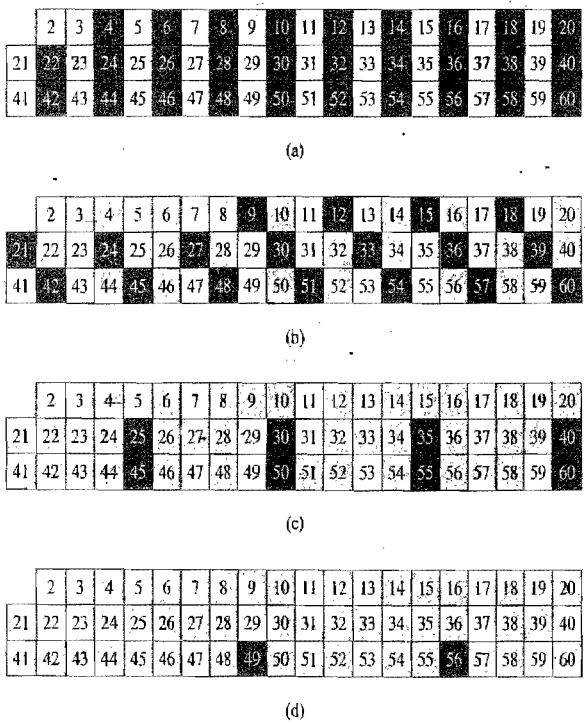


Figure 5.2 The Sieve of Eratosthenes. In this example we are finding all primes less than or equal to 60. (a) Mark all multiples of 2 between 4 and 60, inclusive. (b) The next unmarked value is 3. Mark all multiples of 3 between 9 and 60, inclusive. (c) The next unmarked value is 5. Mark all multiples of 5 between 25 and 60, inclusive. (d) The next unmarked value is 7. Mark all multiples of 7 between 49 and 60. The next unmarked value is 11. Since the square of 11 is 121, and 121 is greater than 60, the algorithm terminates. All remaining unmarked cells represent prime numbers.

numbers. The next prime is 11. The square of its value is 121, which is greater than 60, causing an end to the sieving loop. The unmarked integers that remain are primes.

The Sieve of Eratosthenes is not practical for identifying large prime numbers with hundreds of digits, because the algorithm has complexity $\Theta(n \ln \ln n)$, and n is exponential in the number of digits. However, a modified form of the sieve is still an important tool in number theory research.

When we implement this algorithm in the C programming language, we can use an array of $n - 1$ chars (with indices $0, 1, \ldots, n - 2$) to represent the natural numbers $2, 3, \ldots, n$. The boolean value at index i indicates whether natural number $i + 2$ is marked.

5.3 SOURCES OF PARALLELISM

How should we partition this algorithm? Because the heart of the algorithm is marking elements of the array representing integers, it makes sense to do a domain decomposition, breaking the array into $n - 1$ elements and associating a primitive task with each of these elements.

The key parallel computation is step 3a, where those elements representing multiples of a particular prime k are marked as composite. For the cell representing integer j, this computation is straightforward: if $j \bmod k = 0$, then j is a multiple of k and should be marked.

If a primitive task represents each integer, then two communications are needed to perform step 3b each iteration of the **repeat ... until** loop. A reduction is needed each iteration in order to determine the new value of k, and then a broadcast is needed to inform all the tasks of the new value of k.

Reflecting on this domain decomposition, the good news is that there is plenty of data parallelism to exploit. The bad news is that there are a lot of reduction and broadcast operations.

The next step in our design is to think about how to agglomerate the primitive tasks into more substantial tasks that still allow us to utilize a reasonable number of processors. In the best case we will end up with a new version of the parallel algorithm that requires less computation *and* less communication than the original parallel algorithm.

5.4 DATA DECOMPOSITION OPTIONS

After we agglomerate the primitive tasks, a single task will be responsible for a group of array elements representing several integers. We often call the final grouping of data elements—the result of partitioning, agglomeration, and mapping—the **data decomposition,** or simply "the decomposition."

5.4.1 Interleaved Data Decomposition

First, let's consider an interleaved decomposition of array elements:

- process 0 is responsible for the natural numbers $2, 2+p, 2+2p, \ldots,$
- process 1 is responsible for the natural numbers $3, 3+p, 3+2p, \ldots,$

and so on.

An advantage of the interleaved decomposition is that given a particular array index i, it is easy to determine which process controls that index (process i mod p). A disadvantage of an interleaved decomposition *for this problem* is that it can lead to significant load imbalances among the processes. For example, if two processes are marking multiples of 2, process 0 marks $\lceil (n-1)/2 \rceil$ elements while process 1 marks none. A further disadvantage is that the implementation of step 3b (finding the next prime number) still requires some sort of reduction/broadcast.

5.4.2 Block Data Decomposition

An alternative is a **block data decomposition.** That means we divide the array into p contiguous blocks of roughly equal size. If the number of array elements n is a multiple of the number of processes p, the division is straightforward.

If n is not a multiple of p, then it is more complicated. Suppose $n = 1024$ and $p = 10$. In that case $1024/10 = 102.4$. If we give every process 102 elements, there will be four left over. On the other hand, we cannot give every process 103 elements, because the array is not that large. We cannot simply give the first $p-1$ processes $\lceil n/p \rceil$ combinations and give the last process whatever is left over, because there may not be any elements left (see Exercise 5.2). Allocating no elements to a process is undesirable for two reasons. First, it can complicate the logic of programs in which processes exchange values. Second, it can lead to a less efficient utilization of the communication network.

What we need instead is a block allocation scheme that balances the workload by assigning to each process either $\lceil n/p \rceil$ or $\lfloor n/p \rfloor$ elements. (If n is evenly divisible by p every process will be assigned n/p elements.) Let's consider two different ways of accomplishing this.

The first method begins by computing $r = n \bmod p$. If r is 0, then n is a multiple of p, and every process should get a block of size n/p. If $r > 0$, then the first r processes should get a block of size $\lceil n/p \rceil$ and the remaining $p - r$ processes should get a block of size $\lfloor n/p \rfloor$.

For example, when $n = 1024$ and $p = 10$, the first four processes would get 103 pieces of work, and the last six processes would get 102 pieces of work.

There are two questions we typically need to be able to answer when developing algorithms based on a block allocation of data. What is the range of elements controlled by a particular process? Which process controls a particular element?

Let's answer these questions for our first scheme.

Suppose n is the number of elements and p is the number of processes. The first element controlled by process i is

$$i\lfloor n/p \rfloor + \min(i, r)$$

The last element controlled by process i is the element immediately before the first element controlled by process $i + 1$:

$$(i + 1)\lfloor n/p \rfloor + \min(i + 1, r) - 1$$

The process controlling a particular array element j is

$$\min(\lfloor j/(\lfloor n/p \rfloor + 1) \rfloor, \lfloor (j - r)/\lfloor n/p \rfloor \rfloor)$$

All of these expressions are somewhat complicated. The expressions for the first and last elements controlled by a particular process are not onerous, because each process could compute these values and store the results at the beginning of the algorithm. However, determining the controlling process from the element index would most likely be done on the fly, so the complexity of this expression is worrisome.

The second block allocation scheme we are considering does not concentrate all of the larger blocks among the smaller-numbered processes. Suppose n is the number of elements and p is the number of processes. The first element controlled by process i is

$$\lfloor in/p \rfloor$$

The last element controlled by process i is the element immediately before the first element controlled by process $i + 1$:

$$\lfloor (i + 1)n/p \rfloor - 1$$

The process controlling a particular array element j is

$$\lfloor (p(j + 1) - 1)/n \rfloor$$

Figure 5.3 contrasts these two block data decomposition methods.

The second approach is superior because it requires fewer operations to perform the three most common block management computations, especially since integer division in C automatically rounds down the result. It is the block decomposition method we will use for the remainder of the book.

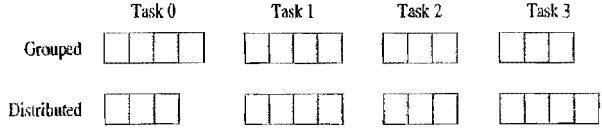


Figure 5.3 An example of two block data decomposition schemes. In this case 14 elements are divided among four tasks. In the first scheme the larger blocks are held by the lowest-numbered tasks; in the second scheme the larger blocks are distributed among the tasks.

5.4.3 Block Decomposition Macros

Let's pause for a moment and define three C macros that can be used in any of our parallel programs where a group of data items is distributed among a set of processors using a block decomposition.

```
#define BLOCK_LOW(id,p,n)  ((id)*(n)/(p))
#define BLOCK_HIGH(id,p,n) (BLOCK_LOW((id)+1,p,n) - 1)
#define BLOCK_SIZE(id,p,n) (BLOCK_LOW((id)+1)-BLOCK_LOW(id))
#define BLOCK_OWNER(index,p,n) (((p)*((index)+1)-1)/(n))
```

Given process rank `id`, number of processes `p`, and number of elements `n`, macro `BLOCK_LOW` expands to an expression whose value is the first, or lowest, index controlled by the process.

Given the same arguments, macro `BLOCK_HIGH` expands to an expression whose value is the last, or highest, index controlled by the process.

With the same three arguments, macro `BLOCK_SIZE` evaluates to the number of elements controlled by process `id`.

Passed an array index, the number of processes, and the total number of array elements, macro `BLOCK_OWNER` evaluates to the rank of the process controlling that element of the array.

These four definitions are the start of a set of utility macros and functions we can reference when constructing our parallel programs.

5.4.4 Local Index versus Global Index

When we decompose an array into pieces distributed among a set of tasks, we must remember to distinguish between the local index of an array element and its global index.

For example, consider an array distributed among tasks as shown in Figure 5.4. Eleven array elements are distributed among three tasks. Each task is responsible for either three or four elements; hence the local indices range from 0 to either 2 or 3. However, each local array represents a portion of the larger, global array, whose indices range from 0 to 10.

We must keep this distinction in mind when transforming sequential programs into parallel programs. Sequential codes always use the global indices to reference array elements. We must substitute the local indices when we write our parallel codes.

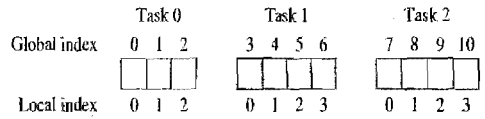


Figure 5.4 When an array is distributed among tasks, you must distinguish between an array element's local index and its global index. Here an 11-element array is distributed blockwise among three tasks.

5.4.5 Ramifications of Block Decomposition

How does our block decomposition affect the implementation of the parallel algorithm?

First, note that the largest prime used to sieve integers up to n is $\sqrt{n}$. If the first process is responsible for integers through $\sqrt{n}$, then finding the next value of k requires no communications at all—it saves a reduction step. Is this assumption reasonable? The first process has about n/p elements. If $n/p > \sqrt{n}$, then it will control all primes through $\sqrt{n}$. Since n is expected to be in the millions, this is a reasonable assumption.

A second advantage of a block decomposition is that it can speed the marking of cells representing multiples of k. Rather than check each array element to see if it represents an integer that is a multiple of k—requiring n/p modulo operations for each prime—the algorithm can find the first multiple of k and then mark that cell (call it j) as well as cells $j+k$, $j+2k$, etc., through the end of the block, for a total of about $(n/p)/k$ assignment statements. In other words, it can use a loop similar to the one used in a sequential implementation of the algorithm. This is much faster.

We have seen, then, how in this case a block decomposition results in fewer computational steps *and* fewer communications steps.

5.5 DEVELOPING THE PARALLEL ALGORITHM

Now that we have determined the data decomposition, we return to the sequential algorithm shown in Figure 5.1 and see how each step translates into equivalent steps in the parallel algorithm.

Step 1 is simple to translate. Instead of a single process creating an entire list of natural numbers, each process in the parallel program will create its portion of the list, containing either $\lceil n/p \rceil$ or $\lfloor n/p \rfloor$ boolean values.

Every process is going to need to know the value of k in order to mark the multiples of k in its region. For that reason, every process in the parallel program executes step 2. This is an example of a parallel program replicating work. Fortunately, in this case the amount of replicated work is trivial.

Step 3a is also easy to translate. Each process is responsible for marking all the multiples of k in its block between k^2 and n. We may need to do a little bit of algebra to determine the location of the first multiple of k in the block, but after that, all we need to do is mark every kth element in the block.

As we have already determined, process 0 is exclusively responsible for determining the next value of k if $p < \sqrt{n}$, which is true for all values of n for which we would reasonably want to execute the parallel algorithm. If process 0 is responsible for finding the next prime in step 3b, which determines the new value of k, then all of the other processes must receive the new value of k so that they may compute the value of the termination expression in the **repeat ... until** loop and possibly use it in the next iteration of the loop.

In other words, we want to copy the up-to-date value of k on process 0 to the local instances of k located on the other processes. This is an example of broadcasting, a global communication function.

5.5.1 Function MPI_Bcast

Let's look at the header of function MPI_Bcast, which enables a process to broadcast one or more data items of the same type to all other processes in a communicator:

```
int MPI_Bcast (
   void         *buffer,   /* Addr of 1st broadcast element */
   int           count,    /* # elements to broadcast */
   MPI_Datatype datatype,  /* Type of elements to broadcast */
   int           root,     /* ID of process doing broadcast */
   MPI_Comm      comm)     /* Communicator */
```

The second parameter, count, indicates how many elements are being broadcast. Every process calling this function needs to specify the same value for count. The first parameter, buffer, is the address of the first data item to be broadcast. The function assumes all of the data items are in contiguous memory locations. The third parameter, datatype, is an MPI constant indicating the type of the data items to be broadcast. Parameter four, root, is the rank of the process broadcasting the data item(s). Finally, the fifth parameter, comm, indicates the communicator, the group of processes participating in this collective communication function.

In the case of our parallel sieve algorithm, process 0 needs to broadcast a single integer, k, to all other processes. Hence the call takes this form:

```
MPI_Bcast (&k, 1, MPI_INT, 0, MPI_COMM_WORLD);
```

After this function has executed, every process has an up-to-date value of k and is able to evaluate the termination condition in the **repeat ... until** loop.

At the conclusion of the **repeat ... until** loop, all the primes between 2 and n have been discovered. They correspond to the unmarked elements of the boolean array. A more meaningful program would then make use of the primes. Since we are more interested in learning about parallel programming than number theory, let's take the easy way out and simply count the number of primes in the range 2 through n.

It is straightforward for each process to count the number of primes (number of array elements equal to 0) in its local array. At that point we need to perform a *sum-reduction* to accumulate these subtotals into a grand total. As we saw in the previous chapter, this is implemented using the MPI function MPI_Reduce.

The task/channel graph for our parallel algorithm appears in Figure 5.5.

5.6 ANALYSIS OF PARALLEL SIEVE ALGORITHM

Now that we have designed a parallel algorithm, let's derive an expression that approximates its execution time.

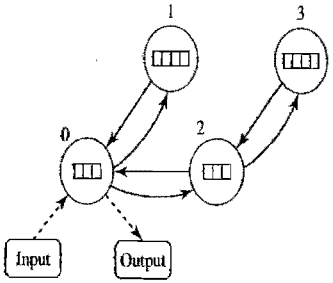


Figure 5.5 Task/channel graph for the parallel Sieve of Eratosthenes algorithm with four tasks. The dotted arrows represent channels used for I/O. The curved arrows represent channels used for the broadcast step. The straight, solid arrows represent channels used for the reduction step (as previously illustrated in Figure 3.12).

Let χ represent the time needed to mark a particular cell as being the multiple of a prime. This time includes not only the time needed to assign 1 to an element of the array, but also time needed for incrementing the loop index and testing for termination. The sequential algorithm has time complexity $\Theta(n \ln \ln n)$. We can determine χ experimentally by running a sequential version of the algorithm. In other words, the expected execution time of the serial algorithm is roughly $\chi n \ln \ln n$.

Since only a single data value is broadcast each iteration, the cost of each broadcast is closely approximated by $\lambda \lceil \log p \rceil$, where λ is message latency.

How many times will this loop iterate? The number of primes between 2 and n is about $n / \ln n$ [11]. Hence a good approximation to the number of loop iterations is $\sqrt{n} / \ln \sqrt{n}$.

Therefore, the expected execution time of the parallel algorithm is approximately

$$\chi(n \ln \ln n)/p + (\sqrt{n} / \ln \sqrt{n})\lambda \lceil \log p \rceil$$

5.7 DOCUMENTING THE PARALLEL PROGRAM

The complete text of the parallel Sieve of Eratosthenes program appears in Figure 5.6. In this section we thoroughly document the program.

We begin with the standard include files. Header file `MyMPI.h` contains macros and function prototypes for the utilities we are developing. From now on, we'll include this header file in our programs. We also define a macro that computes the minimum of two values.

```
/*
 *   Sieve of Eratosthenes
 */

#include <mpi.h>
#include <math.h>
#include <stdio.h>
#include "MyMPI.h"
#define MIN(a,b)  ((a)<(b)?(a):(b))

int main (int argc, char *argv[])
{
   int    count;        /* Local prime count */
   double elapsed_time; /* Parallel execution time */
   int    first;        /* Index of first multiple */
   int    global_count; /* Global prime count */
   int    high_value;   /* Highest value on this proc */
   int    i;
   int    id;           /* Process ID number */
   int    index;        /* Index of current prime */
   int    low_value;    /* Lowest value on this proc */
   char  *marked;       /* Portion of 2,...,'n' */
   int    n;            /* Sieving from 2, ..., 'n' */
   int    p;            /* Number of processes */
   int    proc0_size;   /* Size of proc 0's subarray */
   int    prime;        /* Current prime */
   int    size;         /* Elements in 'marked' */

   MPI_Init (&argc, &argv);

   /* Start the timer */

   MPI_Barrier(MPI_COMM_WORLD);
   elapsed_time = -MPI_Wtime();

   MPI_Comm_rank (MPI_COMM_WORLD, &id);
   MPI_Comm_size (MPI_COMM_WORLD, &p);

   if (argc != 2) {
      if (!id) printf ("Command line: %s <m>\n", argv[0]);
      MPI_Finalize();
      exit (1);
   }

   n = atoi(argv[1]);

   /* Figure out this process's share of the array, as
      well as the integers represented by the first and
      last array elements */

   low_value = 2 + BLOCK_LOW(id,p,n-1);
   high_value = 2 + BLOCK_HIGH(id,p,n-1);
   size = BLOCK_SIZE(id,p,n-1);

   /* Bail out if all the primes used for sieving are
      not all held by process 0 */
```

Figure 5.6 MPI program for Sieve of Eratosthenes.

```
   proc0_size = (n-1)/p;

   if ((2 + proc0_size) < (int) sqrt((double) n)) {
      if (!id) printf ("Too many processes\n");
      MPI_Finalize();
      exit (1);
   }

   /* Allocate this process's share of the array. */

   marked = (char *) malloc (size);

   if (marked == NULL) {
      printf ("Cannot allocate enough memory\n");
      MPI_Finalize();
      exit (1);
   }

   for (i = 0; i < size; i++) marked[i] = 0;
   if (!id) index = 0;
   prime = 2;
   do {
      if (prime * prime > low_value)
         first = prime * prime - low_value;
      else {
         if (!(low_value % prime)) first = 0;
         else first = prime - (low_value % prime);
      }
      for (i = first; i < size; i += prime) marked[i] = 1;
      if (!id) {
         while (marked[++index]);
         prime = index + 2;
      }
      MPI_Bcast (&prime,  1, MPI_INT, 0, MPI_COMM_WORLD);
   } while (prime * prime <= n);
   count = 0;
   for (i = 0; i < size; i++)
      if (!marked[i]) count++;
   MPI_Reduce (&count, &global_count, 1, MPI_INT, MPI_SUM,
      0, MPI_COMM_WORLD);

   /* Stop the timer */

   elapsed_time += MPI_Wtime();


   /* Print the results */

   if (!id) {
      printf ("%d primes are less than or equal to %d\n",
         global_count, n);
      printf ("Total elapsed time: %10.6f\n", elapsed_time);
   }
   MPI_Finalize ();
   return 0;
}
```

Figure 5.6 (contd.) MPI program for Sieve of Eratosthenes.

```
#define MIN(a,b)  ((a)<(b)?(a):(b))
```

The user is supposed to specify the upper range of the sieve as a command-line argument. If this value is missing, we terminate execution. In this case it is vital that each process calls `MPI_Finalize()` before it exits. If the command-line argument exists, we convert the string into an integer.

```
if (argc != 2) {
   if (!id) printf ("Command line: %s <m>\n", argv[0]);
   MPI_Finalize();
   exit (1);
}

n = atoi(argv[1]);
```

The program will find all primes from 2 through n, meaning we are checking the primality of a total of $n - 1$ integers. As we discussed earlier, we will give each process a contiguous block of the array that stores the marks. We determine the low and high values for which this process is responsible, as well as the total number of values it is sieving, using the macros we have developed.

```
low_value = 2 + BLOCK_LOW(id,p,n-1);
high_value = 2 + BLOCK_HIGH(id,p,n-1);
size = BLOCK_SIZE(id,p,n-1);
```

Our algorithm works only if the square of the largest value in process 0's array is greater than the upper limit of the sieve. We add code that checks to ensure that this condition is true. If not, the program terminates.

```
proc0_size = (n-1)/p;

if ((2 + proc0_size) < (int) sqrt((double) n)) {
   if (!id) printf ("Too many processes\n");
   MPI_Finalize();
   exit (1);
}
```

Now we can allocate the process's share of the array. Because a single byte is the smallest unit of memory that can be indexed in C, we declare the array to be of type `char`. If the memory allocation fails, the program terminates.

```
marked = (char *) malloc (size);

if (marked == NULL) {
   printf ("Cannot allocate enough memory\n");
   MPI_Finalize();
   exit (1);
}
```

The elements of the list are unmarked.

```
for (i = 0; i < size; i++) marked[i] = 0;
```

Whew! We have completed Step 1 of the algorithm. Fortunately, the remaining steps can be implemented with much less coding. Step 2, for example, requires only two lines. We will begin by sieving multiples of 2. Integer `prime` is the value of the current prime being sieved. Integer `index` is its index in the array of process 0. We conditionalize the initialization of `index` to process 0 to emphasize that only process 0 uses this variable.

```
if (!id) index = 0;
prime = 2;
```

Now we are at the heart of the program, corresponding to Step 3 in the original algorithm. We implement **repeat ... until** in C as a `do...while` loop.

Each process is responsible for marking in its portion of the list all multiples of `prime` between `prime` squared and `n`. To do this, we need to determine the index corresponding to the first integer needing marking. If `prime` squared is greater than the smallest value stored in the array, then we take the difference between the two values to determine the index of the first element that needs to be marked. Otherwise, we find the remainder when we divide `low_value` by `prime`. If the remainder is 0, `low_value` is a multiple of `prime`, and that is where we should begin marking. Otherwise, we must index into the array to be at the first element that is a multiple of `prime`.

```
if (prime * prime > low_value)
   first = prime * prime - low_value;
else {
   if (!(low_value % prime)) first = 0;
   else first = prime - (low_value % prime);
}
```

The following `for` loop actually does the sieving. Each process marks the multiples of the current prime number from the first index through the end of the array.

```
for (i = first; i < size; i += prime) marked[i] = 1;
```

Process 0 finds the next prime by locating the next unmarked location in the array.

```
if (!id) {
   while (marked[++index]);
   prime = index + 2;
}
```

Process 0 broadcasts the value of the next prime to the other processes.

```
MPI_Bcast (&prime,  1, MPI_INT, 0, MPI_COMM_WORLD);
```

The processes continue to sieve as long as the square of the current prime is less than or equal to the upper limit.

```
} while (prime * prime <= n);
```

Each process counts the number of primes in its portion of the list.

```
count = 0;
for (i = 0; i < size; i++)
   if (!marked[i]) count++;
```

The processes compute the grand total, with the result being stored in variable `global_count` on process 0.

```
MPI_Reduce (&count, &global_count, 1, MPI_INT, MPI_SUM,
   0, MPI_COMM_WORLD);
```

We stop the timer. At this point `elapsed_time` contains the number of seconds it took to execute the algorithm, excluding initial MPI startup time.

```
elapsed_time += MPI_Wtime();
```

Process 0 prints the answer and the elapsed time.

```
if (!id) {
   printf ("%d primes are less than or equal to %d\n",
      global_count, n);
   printf ("Total elapsed time: %10.6f\n", elapsed_time);
}
```

All that remains is a call to `MPI_Finalize` to shut down MPI.

5.8 BENCHMARKING

Let's see how well our model compares with the actual performance of the parallel program finding all primes up to 100 million.

We will execute our parallel program on a commodity cluster of 450 MHz Pentium II CPUs. Each CPU has a fast Ethernet connection to a Hewlett-Packard Procurve 4108GL switch.

First, we determine the value of χ by running a sequential implementation of the program on a single processor of the cluster. The sequential program executes in 24.900 seconds. Hence

$$\chi = \frac{24.900 \text{ sec}}{100{,}000{,}000 \ln \ln 100{,}000{,}000} = 85.47 \text{ nanoseconds}$$

We also need to determine λ. By performing a series of broadcasts on $2, \ldots, 8$ processors, we determine $\lambda = 250$ μsec.

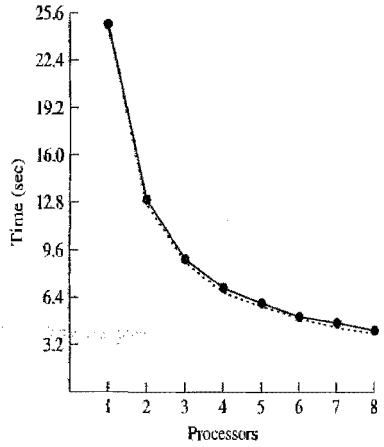


Figure 5.7 Comparison of the predicted (dotted line) and actual (solid line) execution times of the parallel Sieve of Eratosthenes program.

Plugging these values into our formula for the expected execution time of the parallel algorithm, we find

$$\chi(n \ln \ln n)/p + (\sqrt{n}/\ln \sqrt{n})\lambda \log p = 24.900/p + 0.2714\lceil \log p \rceil \text{ sec}$$

We benchmark our parallel program by executing it 40 times—five times for each number of processors between 1 and 8. For each number of processors we compute the mean execution time. Figure 5.7 compares our experimental results with the execution times predicted by our model. The average error of the predictions for $2, \ldots, 8$ processors is about 4 percent.

5.9 IMPROVEMENTS

While the parallel sieve algorithm we have developed does exhibit good performance, there are a few modifications to the program that can improve performance significantly. In this section we present three modifications to the parallel sieve algorithm. Each change builds on the previous ones.

5.9.1 Delete Even Integers

Since 2 is the only even prime, there is little sense in setting aside half of the boolean values in the array for even integers. Changing the sieve algorithm so that only odd integers are represented halves the amount of storage required and doubles the speed at which multiples of a particular prime are marked. With this change the estimated execution time of the sequential algorithm becomes

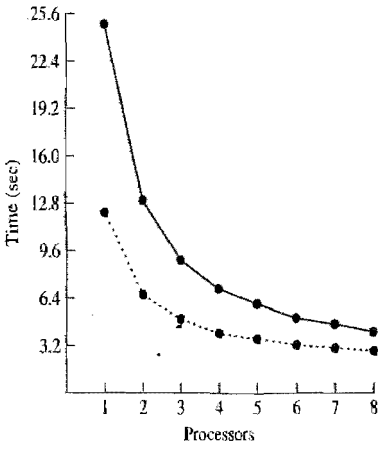


Figure 5.8 Execution time of the original (solid line) and improved (dotted line) parallel programs performing the Sieve of Eratosthenes.

approximately

$$\chi(n \ln \ln n)/2$$

and the estimated execution time of the parallel algorithm becomes approximately

$$\chi(n \ln \ln n)/(2p) + (\sqrt{n}/\ln \sqrt{n})\lambda \log p$$

Figure 5.8 plots the results of benchmarking the original parallel sieve algorithm and the improved algorithm sieving 100 million integers on $1, 2, \ldots, 8$ processors. As expected, the time required for the improved algorithm is about half the time required for the original algorithm, at least when the number of processors is small.

In fact, while our improved sieve runs twice as fast as our original program on one processor, it executes only slightly faster on eight processors. The computation time of the improved program is significantly lower than that of the original program, but the communication requirements are identical. As the number of processors increases, the relative importance of the communication component to overall execution time grows, shrinking the difference between the two programs.

5.9.2 Eliminate Broadcast

Consider step 3b of the original algorithm, in which the new sieve value k is identified. We made this step parallel by letting one process identify the new value of k and then broadcast it to the other processes. During the course of the program's execution this broadcast step is repeated about $\sqrt{n}/\ln \sqrt{n}$ times.

Why not let every task identify the new value of k? In our original data decomposition scheme this is impossible, because only task 0 controls the array elements associated with the integers $2, 3, \ldots, \sqrt{n}$. What if we chose to replicate these values?

Suppose in addition to each task's set of about n/p integers, each task also has a separate array containing integers $3, 5, 7, \ldots, \lfloor\sqrt{n}\rfloor$. Before finding the primes from 3 through n, each task will use the sequential algorithm to find the primes from 3 through $\lfloor\sqrt{n}\rfloor$. Once this has been done, each task now has its own private copy of an array containing all the primes between 3 and $\lfloor\sqrt{n}\rfloor$. Now the tasks can sieve their portions of the larger array without any broadcast steps.

Eliminating the broadcast step improves the speed of the parallel algorithm if

$$\begin{aligned} & (\sqrt{n}/\ln\sqrt{n})\lambda\lceil\log p\rceil > \chi\sqrt{n}\ln\ln\sqrt{n} \\ \Rightarrow\ & (\lambda\lceil\log p\rceil)/\ln\sqrt{n} > \chi\ln\ln\sqrt{n} \\ \Rightarrow\ & \lambda > \chi\ln\ln\sqrt{n}\ln\sqrt{n}/\lceil\log p\rceil \end{aligned}$$

The expected time complexity of the parallel algorithm is now approximately

$$\chi\left((n\ln\ln n)/(2p) + \sqrt{n}\ln\ln\sqrt{n}\right) + \lambda\lceil\log p\rceil$$

(The final term represents the time needed to do the sum-reduction.)

5.9.3 Reorganize Loops

For much of the execution of the parallel sieve algorithm, each process is marking widely dispersed elements of a very large array, leading to a poor cache hit rate. Think of the heart of the algorithm developed in the previous subsection as two large loops. The outer loop iterates over prime sieve values between 3 and $\lfloor\sqrt{n}\rfloor$, while the inner loop iterates over the process's share of the integers between 3 and n. If we exchange the inner and outer loops, we can improve the cache hit rate. We can fill the cache with a section of the larger subarray, then strike all the multiples of all the primes less than $\lfloor\sqrt{n}\rfloor$ on that section before bringing in the next section of the subarray. (See Figure 5.9.)

5.9.4 Benchmarking

Figure 5.10 plots the execution times of the original parallel Sieve of Eratosthenes program and all three improved versions, when finding primes less than 100 million on $1, 2, \ldots, 8$ processors. The underlying hardware is a commodity cluster consisting of 450-MHz Pentium II CPUs connected by fast Ethernet to a Hewlett-Packard Procurve 4108GL switch.

The execution time of the original sequential program is the same as Sieve 1 on one processor. On eight processors, our parallel sieve program that incorporates every described optimization executes about 72.8 times faster than the original sequential program. The larger share of the increase (a factor of 9.8) results from

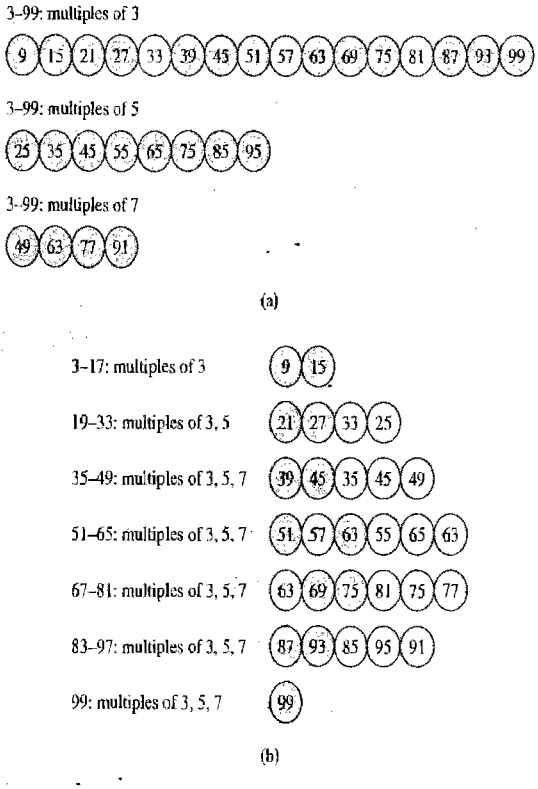


Figure 5.9 Changing the order in which composite integers are marked can dramatically improve the cache hit rate. In this example we are finding primes between 3 and 99. Suppose the cache has four lines, and each line can hold four bytes. One line contains bytes representing integers 3, 5, 7, and 9; the next line holds bytes representing 11, 13, 15, and 17; etc. (a) Sieving all multiples of one prime before considering next prime. Shaded circles represent cache misses. By the time the algorithm returns to the bytes representing smaller integers, they are no longer in the cache. (b) Sieving multiples of all primes for 8 bytes in two cache lines before considering the next group of 8 bytes. Fewer shaded circles indicates the cache hit rate has improved.

eliminating the storage and manipulation of even integers and inverting the two principal loops to improve the cache hit rate. The smaller share of the increase (a factor of about 7.4) results from redundantly computing primes up to $\sqrt{n}$ to eliminate broadcasts and using eight processors instead of one. Our greater gains, then, were the result of improvements to the sequential algorithm before parallelism was applied.

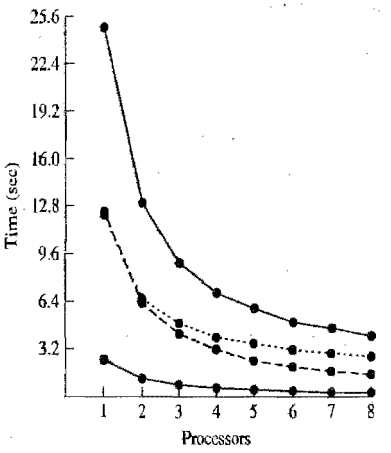


Figure 5.10 Execution time of four parallel implementations of the Sieve of Eratosthenes on a commodity cluster of 450 MHz Pentium II processors connected by fast Ethernet. The upper solid line is the original program. The dotted line is the execution time of the program that does not store or strike even integers. The dashed line plots the execution time of the program that eliminates broadcasts. The lower solid line shows the execution time of the program that incorporates the additional optimization of interchanging loops in order to improve the cache hit rate.

5.10 SUMMARY

We began with a sequential algorithm for the Sieve of Eratosthenes and used the domain decomposition methodology to identify parallelism. For this algorithm, a blockwise distribution of array values to processes is superior to an interleaved distribution. The data-parallel algorithm we designed requires that task 0 broadcasts the current prime to the other tasks. The resulting parallel program uses the function `MPI_Bcast` to perform this broadcast operation. The program achieves good performance on a commodity cluster finding primes up to 100 million.

We then examined three improvements to the original parallel version. The first improvement eliminated all manipulation of even integers, roughly cutting in half both storage requirements and overall execution time. The second improvement eliminates the need for a broadcast step by making redundant the portion of the computation that determines the next prime. The cost of this improvement is a requirement that each process store all odd integers between 3 and $\sqrt{n}$.

The third enhancement improved the cache hit rate by striking all composite values for a single cache-full of integers before moving on to the next segment.

Note that our fourth program executes faster on one processor than our original program does on eight processors. Comparing both programs on eight processors, the fourth program executes more than 11 times faster than our original program. It is important to maximize single processor performance even when multiple processors are available.

5.11 KEY TERMS

block decomposition data decomposition

5.12 BIBLIOGRAPHIC NOTES

Luo [76] presents a version of the Sieve of Eratosthenes in which neither multiples of 2 nor multiples of 3 appear in the array of integers to be marked.

5.13 EXERCISES

5.1 Consider a simple block allocation of n data items to p processes in which the first $p-1$ processes get $\lceil n/p \rceil$ items each and the last process gets what is left over.

a. Find values for n and p where the last process does not get any elements.

b. Find values for n and p where $\lfloor p/2 \rfloor$ processes do not get any values. Assume $p > 1$.

5.2 This chapter presents two block data decomposition strategies that assign n elements to p processes such that each process is assigned either $\lfloor n/p \rfloor$ or $\lceil n/p \rceil$ elements. For each pair of values n and p, use a table or an illustration to show how these two schemes would assign array elements to processes:

a. $n = 15$ and $p = 4$

b. $n = 15$ and $p = 6$

c. $n = 16$ and $p = 5$

d. $n = 18$ and $p = 4$

e. $n = 20$ and $p = 6$

f. $n = 23$ and $p = 7$

5.3 Use the analytical model developed in Section 5.6 to predict the execution time of the original parallel sieve program on $1, 2, \ldots, 16$ processors. Assume $n = 10^8$, $\lambda = 250$ μsec, and $\chi = 0.0855$ μsec.

5.4 Use the analytical model developed in Section 5.9.1 to predict the execution time of the second version of the parallel sieve program (the one that does not store or mark even integers). Compare the execution

time predicted by the model to the actual execution time reported in column 2 of Table 5.1. What is the average error of the predictions for $2, \ldots, 8$ processors?

Table 5.1 Mean execution times (in seconds) of four parallel implementations of the Sieve of Eratosthenes on a commodity cluster of 450 MHz Pentium II processors connected by fast Ethernet. Sieve 1 is the original program. Sieve 2 does not store or strike even integers. Sieve 3 incorporates the additional optimization of eliminating broadcasts by computing primes between 2 and $\sqrt{n}$ on each processor. Sieve 4 incorporates the additional optimization of interchanging loops in order to improve the cache hit rate.

Processors	Sieve 1	Sieve 2	Sieve 3	Sieve 4
1	24.900	12.237	12.466	2.543
2	12.721	6.609	6.378	1.330
3	8.843	5.019	4.272	0.901
4	6.768	4.072	3.201	0.679
5	5.794	3.652	2.559	0.543
6	4.964	3.270	2.127	0.456
7	4.371	3.059	1.820	0.391
8	3.927	2.856	1.585	0.342

5.5 Use the analytical model developed in Section 5.9.2 as a starting point to predict the execution time of the third version of the parallel sieve program. Assume $n = 10^8$, $\lambda = 250$ μsec, and $\chi = 0.0855$ μsec. Compare the execution time predicted by your model to the actual execution time reported in column *Sieve 3* of Table 5.1. What is the average error of the predictions for $2, \ldots, 8$ processors?

5.6 Modify the parallel Sieve of Eratosthenes program presented in the text to incorporate the first improvement described in Section 5.9: it should not set aside memory for even integers. Benchmark your program, comparing its performance with that of the original parallel sieve program.

5.7 Modify the parallel Sieve of Eratosthenes program presented in the book to incorporate the first two improvements described in Section 5.9. Your program should not set aside memory for even integers, and each process should use the sequential Sieve of Eratosthenes algorithm on a separate array to find all primes between 3 and $\lfloor \sqrt{n} \rfloor$. With this information, the call to `MPI_Bcast` can be eliminated. Benchmark your program, comparing its performance with that of the original parallel sieve program.

5.8 Modify the parallel Sieve of Eratosthenes program presented in the text to incorporate all three improvements described in Section 5.9. Benchmark your program, comparing its performance with that of the original parallel sieve program.

5.9 All the parallel sieve algorithms developed in this chapter are the result of a domain decomposition of the original algorithm. Write a parallel

Sieve of Eratosthenes program based upon a functional decomposition of the algorithm. Suppose there are p processes finding primes up to n. (The program gets these parameters from the command line.) In the first step each process independently identifies primes up to $\sqrt{n}$. In step two each process sieves the list of integers with $1/p$th of the primes between 2 and $\sqrt{n}$. During the third step the processes OR-reduce their arrays into a single array held by process 0. In the last step process 0 counts the unmarked elements of the array and prints the prime number count.

For example, suppose three processes are cooperating to find primes up to 1000. Each process allocates an array of 999 elements, representing the integers 2 through 1000. Each process identifies the primes less than or equal to $\sqrt{1000}$: 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31. Process 0 sieves its array with the primes 2, 7, 17, and 29; process 1 sieves its array with the primes 3, 11, 19, and 31; and process 2 sieves its array with the primes 5, 13, and 23.

5.10 Identify three disadvantages of the parallel program design described in the previous exercise compared to the original parallel design described in this chapter.

5.11 The simplest harmonic progression is

$$\frac{1}{1}, \frac{1}{2}, \frac{1}{3}, \ldots$$

Let $S_n = \sum_{i=1}^{n} 1/i$.

a. Write a parallel program that computes these sums to arbitrary precision after the decimal point. For example, $S_7 = 2.592857142857$, to 12 digits of precision after the decimal point. Process 0 should query the user for the two parameters, n and d, and broadcast these parameters to the other processes. Processes should work together to compute S_n to d digits of precision after the decimal point. After S_n has been computed, process 0 should print its value.

b. Benchmark the program computing $S_{1,000,000}$ to 100 digits of precision, using various numbers of processors.

CHAPTER

6

Floyd's Algorithm

Not once or twice in our rough island story
The path of duty was the path of glory.
Alfred, Lord Tennyson, ***Ode on the Death of the Duke of Wellington***

6.1 INTRODUCTION

Travel maps often contain tables showing the driving distances between pairs of cities. At the intersection of the row representing city A and the column representing city B is a cell containing the length of the shortest path of roads from A to B. In the case of longer trips, this route most likely passes through other cities represented in the table. Floyd's algorithm is a classic method for generating this kind of table.

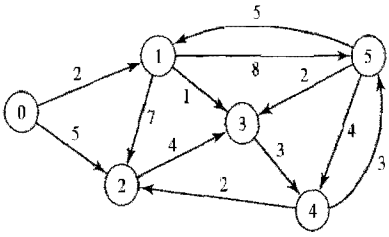
In this chapter we will design, analyze, program, and benchmark a parallel version of Floyd's algorithm. We will begin to develop a suite of functions that can read matrices from files and distribute them among MPI processes, as well as gather matrix elements from MPI processes and print them.

This chapter discusses the following MPI functions:

- `MPI_Send`, which allows a process to send a message to another process
- `MPI_Recv`, which allows a process to receive a message sent by another process

6.2 THE ALL-PAIRS SHORTEST-PATH PROBLEM

A **graph** is a set consisting of V, a finite set of vertices, and E, a finite set of edges between pairs of vertices. Figure 6.1a is a pictorial representation of a graph,



(a)

	0	1	2	3	4	5
0	0	2	5	∞	∞	∞
1	∞	0	7	1	∞	8
2	∞	∞	0	4	∞	∞
3	∞	∞	∞	0	3	∞
4	∞	∞	2	∞	0	3
5	∞	5	∞	2	4	0

(b)

	0	1	2	3	4	5
0	0	2	5	3	6	9
1	∞	0	6	1	4	7
2	∞	15	0	4	7	10
3	∞	11	5	0	3	6
4	∞	8	2	5	0	3
5	∞	5	6	2	4	0

(c)

Figure 6.1 (a) A weighted, directed graph. (b) Representation of the graph as an adjacency matrix. Element (i, j) represents the length of the edge from i to j. Nonexistent edges are considered to have infinite length. (c) Solution to the all-pairs shortest path problem. Element (i, j) represents the length of the shortest path from vertex i to vertex j. The infinity symbol represents nonexistent paths.

in which vertices appear as labeled circles and edges appear as lines between pairs of circles. To be more precise, Figure 6.1a is a picture of a weighted, directed graph. It is a **weighted graph** because a numerical value is associated with each edge. Weights on edges can have a variety of meanings. In the case of shortest path problems, edge weights correspond to distances. It is a **directed graph** because every edge has an orientation (represented by an arrowhead).

Given a weighted, directed graph, the **all-pairs shortest-path problem** is to find the length of the shortest path between every pair of vertices. The length of a path is strictly determined by the weights of its edges, not the number of edges traversed. For example, the length of the shortest path between vertex 0 and vertex 5 in Figure 6.1a is 9; it traverses four edges ($0 \rightarrow 1$, $1 \rightarrow 3$, $3 \rightarrow 4$, and $4 \rightarrow 5$).

If we are going to solve this problem on a computer, we must find a convenient way to represent a weighted, directed graph. The **adjacency matrix** is the data structure of choice for this application, because it allows constant-time access to every edge and does not consume more memory than is required for storing the solution. An adjacency matrix is an $n \times n$ matrix representing a graph with n vertices. In the case of a weighted graph, the value of matrix element (i, j) is the weight of the edge from vertex i to vertex j. Depending upon the application, the way that nonexistent edges are represented varies. In the case of the single-source shortest-path problem, nonexistent edges are assigned extremely high values (such as the maximum integer representable by the underlying architecture). For convenience, we will use the symbol ∞ to represent this extremely high value. Figure 6.1b is an adjacency matrix representation of the same graph shown pictorially in Figure 6.1a.

Floyd's Algorithm:

```
Input:  n — number of vertices
        a[0..n - 1, 0..n - 1] — adjacency matrix
Output: Transformed a that contains the shortest path lengths

for k ← 0 to n - 1
  for i ← 0 to n - 1
    for j ← 0 to n - 1
      a[i, j] ← min(a[i, j], a[i, k] + a[k, j])
    endfor
  endfor
endfor
```

Figure 6.2 Floyd's algorithm is an $\Theta(n^3)$ time algorithm that solves the all-pairs shortest-path problem. It transforms an adjacency matrix into a matrix containing the length of the shortest path between every pair of vertices.

When the algorithm terminates, the matrix contains the lengths of the shortest path between every pair of vertices. Figure 6.1c is the solution of the all-pairs shortest-path problem for the graph represented in Figure 6.1a.

More than 40 years ago Floyd invented an $\Theta(n^3)$ time algorithm for solving the all-pairs shortest-path problem. Floyd's algorithm appears in Figure 6.2. For more information on this algorithm, see Cormen et al. [18].

6.3 CREATING ARRAYS AT RUN TIME

A program manipulating an array is more useful if the size of the array can be specified at run-time, because it does not have to be recompiled when the size of the array to be manipulated changes. Allocating a one-dimensional array in C is easily done by declaring a scalar pointer and allocating memory from the heap with a malloc statement. For example, here is a way to allocate matrix A, a one-dimensional, n-element array of integers:

```
int *A;
...
A = (int *) malloc (n * sizeof(int));
```

Allocating a two-dimensional array is more complicated, however, since C treats a two-dimensional array as an array of arrays. We want to ensure that the array elements occupy contiguous memory locations, so that we can send or receive the entire contents of the array in a single message.

Here is one way to allocate a two-dimensional array (see Figure 6.3). First, we allocate the memory where the array values are to be stored. Second, we allocate the array of pointers. Third, we initialize the pointers.

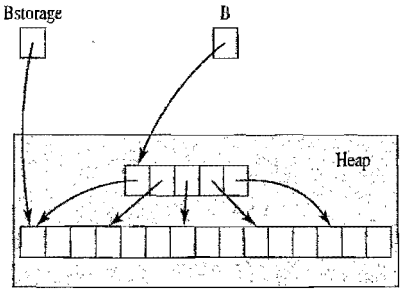


Figure 6.3 Allocating a 5 x 3 matrix is a three-step process. First, the memory for the 15 matrix values is allocated from the heap. Variable `Bstorage` points to the start of this block of memory. Second, the memory for the five row pointers is allocated from the heap. Variable `B` points to the start of this block of memory. Third, the values of the pointers `B[0]`, `B[1]`,..., `B[4]` are initialized.

For example, the following C code allocates B, a two-dimensional array integers. The array has m rows and n columns:

```
int **B, *Bstorage, i;
...
Bstorage = (int *) malloc (m * n * sizeof(int));
B = (int **) malloc (m * sizeof(int *));
for (i = 0; i < m; i++)
   B[i] = &Bstorage[i*n];
```

The elements of B may be initialized in various ways. If they are initializ through a series of assignment statements referencing `B[0][0]`, `B[0][1]`, et there is little room for error. However, if the elements of `B` are initialized en mass for example, through a function call that reads the matrix elements from a fi remember to use `Bstorage`, rather than `B`, as the starting address.

6.4 DESIGNING THE PARALLEL ALGORITHM

6.4.1 Partitioning

Our first step is to determine whether to choose a domain decomposition a functional decomposition. In this case, the choice is obvious. Looking at t pseudocode in Figure 6.2, we see that the algorithm executes the same assignme statement n^3 times. Unless we subdivide this statement, there is no functio parallelism. In contrast, it's easy to perform a domain decomposition. We c

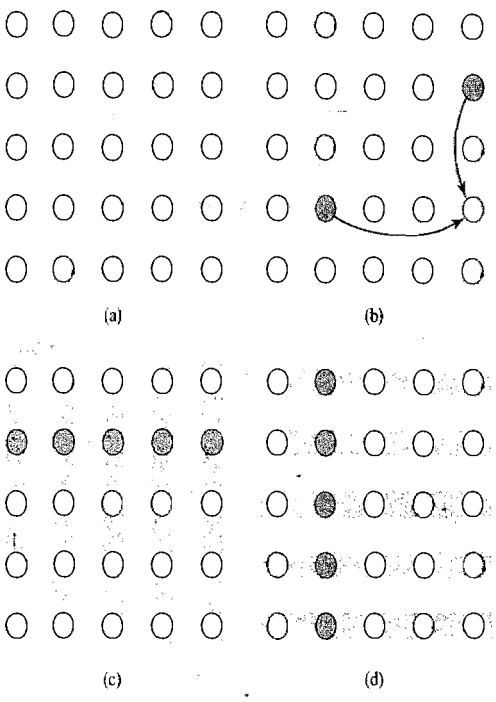


Figure 6.4 Partitioning and communication in Floyd's algorithm. (a) A primitive task is associated with each element of the distance matrix. (b) Updating $a[3, 4]$ when $k = 1$. The new value of $a[3, 4]$ depends upon its previous value and the values of $a[3, 1]$ and $a[1, 4]$. (c) During iteration k every task in row k must broadcast its value to the other tasks in the same column. In this drawing $k = 1$. (d) During iteration k every task in column k must broadcast its value to the other tasks in the same row. In this drawing $k = 1$.

divide matrix A into its n^2 elements and associate a primitive task with each element (Figure 6.4a).

6.4.2 Communication

Each update of element $a[i, j]$ requires access to elements $a[i, k]$ and $a[k, j]$. For example, Figure 6.4b illustrates the elements needed to update $a[3, 4]$ when $k = 1$. Notice that for any particular value of k, element $a[k, m]$ is needed by every task associated with elements in column m. Similarly, for any particular value of k, element $a[m, k]$ is needed by every task associated with elements in row m. What this means is that during iteration k each element in row k of a gets

broadcast to the tasks in the same column (Figure 6.4c). Likewise, each element in column k of a gets broadcast to the tasks in the same row (Figure 6.4d).

It's important to question whether every element of a can be updated simultaneously. After all, if updating $a[i, j]$ requires the values of $a[k, k]$ and $a[k, j]$, shouldn't we have to compute those values first?

The answer to this question is no. The reason is that the values of $a[i, k]$ and $a[k, j]$ don't change during iteration k. That's because during iteration k the update to $a[i, k]$ takes this form:

$$a[i, k] \leftarrow \min(a[i, k], a[i, k] + a[k, j])$$

Since all values are positive, $a[i, k]$ can't decrease. Similarly, the update to $a[k, j]$ takes this form:

$$a[k, j] \leftarrow \min(a[k, j], a[k, k] + a[k, j])$$

The value of $a[k, j]$ can't decrease. Hence there is no dependence between the update of $a[i, j]$ and the updates of $a[i, k]$ and $a[k, j]$. In short, for each iteration k of the outer loop, we can perform the broadcasts and then update every element of a in parallel.

6.4.3 Agglomeration and Mapping

We'll use the decision tree of Figure 3.7 to determine our agglomeration and mapping strategy. The number of tasks is static, the communication pattern among tasks is structured, and the computation time per task is constant. Hence we should agglomerate tasks to minimize communication, creating one task per MPI process.

Our goal, then, is to agglomerate n^2 primitive tasks into p tasks. How should we collect them? Two natural agglomerations group tasks in the same row or column (Figure 6.5). Let's examine the consequences of both of these agglomerations.

If we agglomerate tasks in the same row, the broadcast that occurs among primitive tasks in the same row (Figure 6.4d) is eliminated, because all of these data values are local to the same task. With this agglomeration, during every iteration of the outer loop one task will broadcast n elements to all the other tasks. Each broadcast requires time $\lceil \log p \rceil (\lambda + n/\beta)$.

If we agglomerate tasks in the same column, then the broadcast that occurs among primitive tasks in the same column (Figure 6.4c) is eliminated. This agglomeration, too, results in a message passing time of $\lceil \log p \rceil (\lambda + n/\beta)$ per iteration.

(The truth is that we haven't considered an even better agglomeration, which groups primitive tasks associated with $(n/\sqrt{p}) \times (n/\sqrt{p})$ blocks of elements of A. We'll develop a matrix-vector multiplication program based on this data decomposition in Chapter 8, when we have a lot more MPI functions under our belt.)

To decide between the rowwise and columnwise agglomerations, we need to look outside the computational kernel of the algorithm. The parallel program must input the distance matrix from a file. Assume that the file contains the matrix

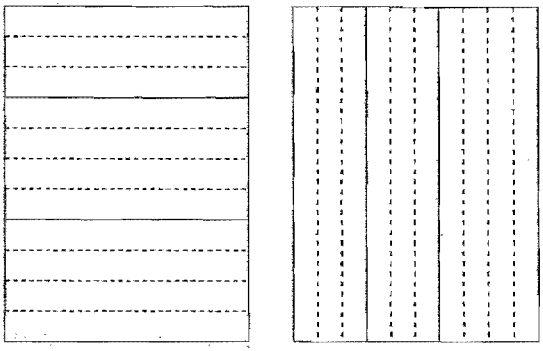


Figure 6.5 Two data decompositions for matrices. (a) In a rowwise block-striped decomposition, each process is responsible for a contiguous group of rows. Here 11 rows are divided among three processes. (b) In a columnwise block-striped decomposition, each process is responsible for a contiguous group of columns. Here 10 columns are divided among three processes.

in row-major order. (The file begins with the first row, then the second row, etc.) In C, matrices are also stored in primary memory in row-major order. Hence distributing rows among processes is much easier if we choose a rowwise block-striped decomposition. This distribution also makes it much simpler to output the result matrix in row-major order. For this reason we choose the rowwise block-striped decomposition.

6.4.4 Matrix Input/Output

We must now decide how we are going to support matrix input/output.

First, let's focus on reading the distance matrix from a file. We could have each process open the file, seek to the proper location in the file, and read its portion of the adjacency matrix. However, we will let one process be responsible for file input. Before the computational loop, this process will read the matrix and distribute it to the other processes. Suppose we have p processes. If process $p-1$ is responsible for reading and distributing the matrix elements, it is easy to implement the program so that no extra space is allocated for file input buffering.

Here is the reason why. If process i is responsible for rows $\lfloor in/p \rfloor$ through $\lfloor (i+1)n/p \rfloor - 1$, then process $p-1$ is responsible for $\lceil n/p \rceil$ rows (see Exercise 6.1). That means no process is responsible for more rows than process $p-1$. Process $p-1$ can use the memory that will eventually store its $\lceil n/p \rceil$ rows to buffer the rows it inputs for the other processes.

Figure 6.6 shows how this method works. The last process opens the file, reads the rows destined for process 0, and sends these rows to process 0. It repeats these steps for the other processes. Finally, it reads the rows it is responsible for.

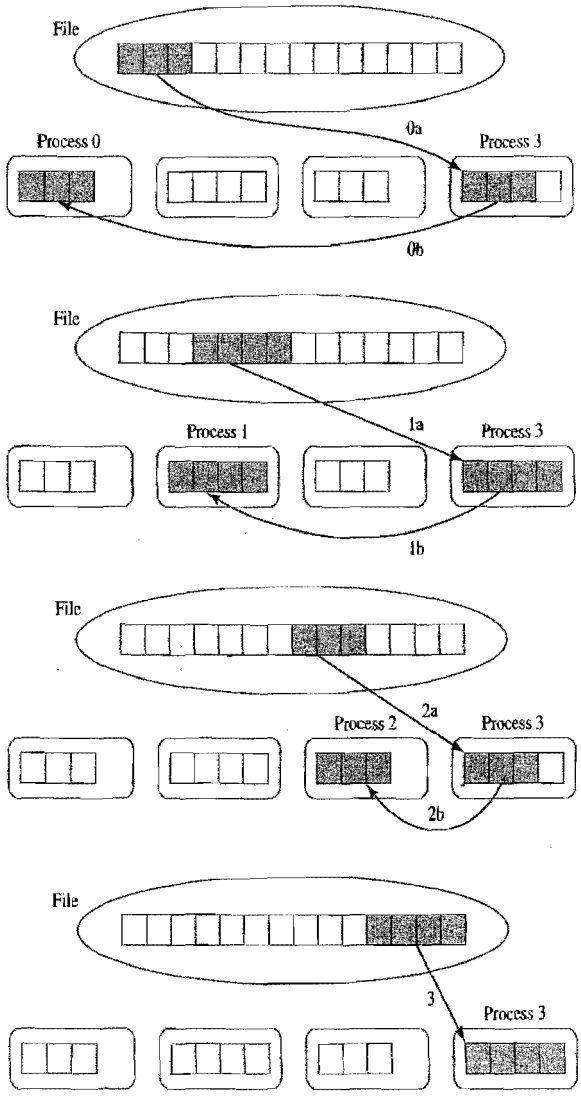


Figure 6.6 Example of a single process managing file input. Here there are four processes, labeled 0, 1, 2, and 3. Process 3 opens the file for reading. In step 0a it reads process 0's share of the data; in step 0b it passes the data to process 0. In steps 1 and 2 it does the same for processes 1 and 2, respectively. In step 3 it inputs its own data.

The complete function, called `read_row_striped_matrix`, appears in Appendix B. Given the name of the input file, the data type of the matrix elements, and a communicator, it returns (1) a pointer to an array of pointers, allowing the matrix elements to be accessed via double-subscripting, (2) a pointer to the location containing the actual matrix elements, and (3) the dimensions of the matrix.

Our implementation of Floyd's algorithm will print the distance matrix twice: when it contains the original set of distances and after it has been transformed into the shortest-path matrix.

Process 0 does all the printing to standard output, so we can be sure the values appear in the correct order. First it prints its own submatrix, then it calls upon each of the other processes in turn to send their submatrices. Process 0 will receive each submatrix and print it.

Little is required of processes $1, 2, \ldots, p-1$. Each of these processes simply waits for a message from process 0, then sends process 0 its portion of the matrix.

Using this protocol, we ensure that process 0 never receives more than one submatrix at a time. Why don't we just let every process fire its submatrix to process 0? After all, process 0 can distinguish between them by specifying the rank of the sending process in its call to `MPI_Recv`. The reason we don't let processes send data to process 0 until requested is we don't want to overwhelm the processor on which process 0 is executing. There is only a finite amount of bandwidth into any processor. If process 0 needs data from process 1 in order to proceed, we don't want the message from process 1 to be delayed because messages are also being received from many other processes.

The source code for function `print_row_striped_matrix` appears in Appendix B.

6.5 POINT-TO-POINT COMMUNICATION

In our function that reads the matrix from a file, process $p-1$ reads a contiguous group of matrix rows, then sends a message containing these rows directly to the process responsible for managing them. In our function that prints the matrix, each process (other than process 0) sends process 0 a message containing its group of matrix rows. Process 0 receives each of these messages and prints the rows to standard output. These are examples of point-to-point communications.

A **point-to-point communication** involves a pair of processes. In contrast, the collective communication operations we have previously explored involve every process in a group.

Figure 6.7 illustrates a point-to-point communication. In this example, process h is not involved in a communication. It continues executing statements manipulating its local variables. Process i performs local computations, then sends a message to process j. After the message is sent, it continues on with its computation. Process j performs local computations, then blocks until it receives a message from process i.

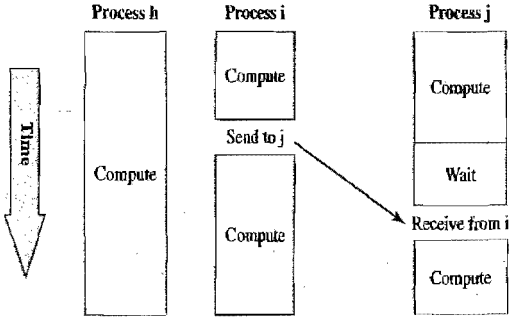


Figure 6.7 Point-to-point communications involve pairs of processes.

```
...
if (id == i) {
   ...
   /* Send message to j */
   ...
} else if (id == j) {
   ...
   /* Receive message from i */
   ...
}
...
```

Figure 6.8 MPI functions performing point-to-point communications often occur inside conditionally executed code.

If every MPI process executes the same program, how can one process send a message while a second process receives a message and a third process does neither?

In order for execution of MPI function calls to be limited to a subset of the processes, these calls must be inside conditionally executed code. Figure 6.8 demonstrates one way that process i could send a message to process j, while the remaining processes skip the message-passing function calls.

Now let's look at the headers of two MPI functions that we can use to perform a point-to-point communication.

6.5.1 Function `MPI_Send`

The sending process calls function `MPI_Send`:

```
int MPI_Send (
      void         *message,
      int           count,
```

```
    MPI_Datatype  datatype,
    int           dest,
    int           tag,
    MPI_Comm      comm
)
```

The first parameter, `message`, is the starting address of the data to be transmitted. The second parameter, `count`, is the number of data items, while the third parameter, `datatype`, is the type of the data items. All of the data items must be of the same type. Parameter 4, `dest`, is the rank of the process to receive the data. The fifth parameter, `tag`, is an integer "label" for the message, allowing messages serving different purposes to be identified. Finally, the sixth parameter, `comm`, indicates the communicator in which this message is being sent.

Function `MPI_Send` blocks until the message buffer is once again available. Typically the run-time system copies the message into a system buffer, enabling `MPI_Send` to return control to the caller. However, it does not have to do this.

6.5.2 Function `MPI_Recv`

The receiving process calls function `MPI_Recv`:

```
int MPI_Recv (
    void          *message,
    int           count,
    MPI_Datatype  datatype,
    int           source,
    int           tag,
    MPI_Comm      comm,
    MPI_Status    *status
)
```

The first parameter, `message`, is the starting address where the received data is to be stored. Parameter 2, `count`, is the maximum number of data items the receiving process is willing to receive, while parameter 3, `datatype`, is the type of the data items. The fourth parameter, `source`, is the rank of the process sending the message. The fifth parameter, `tag`, is the desired tag value for the message. Parameter 6, `comm`, identifies the communicator in which this message is being passed.

Note the seventh parameter, `status`, which appears in `MPI_Recv`, but not `MPI_Send`. Before calling `MPI_Recv`, you need to allocate a record of type `MPI_Status`. Parameter `status` is a pointer to this record, which is the only user-accessible MPI data structure.

Function `MPI_Recv` blocks until the message has been received (or until an error condition causes the function to return). When function `MPI_Recv`

returns, the status record contains information about the just-completed function. In particular:

- `status->MPI_SOURCE` is the rank of the process sending the message.
- `status->MPI_TAG` is the message's tag value.
- `status->MPI_ERROR` is the error condition.

Why would you need to query about the rank of the process sending the message or the message's tag value, if these values are specified as arguments to function `MPI_Recv`? The reason is that you have the option of indicating that the receiving process should receive a message from *any* process by making the constant `MPI_ANY_SOURCE` the fourth argument to the function, instead of a process number. Similarly, you can indicate that the receiving process should receive a message with any tag value by making the constant `MPI_ANY_TAG` the fifth argument to the function. In these circumstances, it may be necessary to look at the status record to find out the identity of the sending process and/or the value of the message's tag.

6.5.3 Deadlock

"A process is in a deadlock state if it is blocked waiting for a condition that will never become true" [3]. It is not hard to write MPI programs with calls to `MPI_Send` and `MPI_Recv` that cause processes to deadlock.

For example, consider two processes with ranks 0 and 1. Each wants to compute the average of `a` and `b`. Process 0 has an up-to-date value of `a`; process 1 has an up-to-date value of `b`. Process 0 must read `b` from 1; while process 1 must read `a` from 0. Consider this implementation:

```
float       a, b, c;
int         id;                 /* Process rank */
MPI_Status status;
...
if (id == 0) {
   MPI_Recv (&b, 1, MPI_FLOAT, 1, 0, MPI_COMM_WORLD, &status);
   MPI_Send (&a, 1, MPI_FLOAT, 1, 0, MPI_COMM_WORLD);
   c = (a + b) / 2.0;
} else if (id == 1) {
   MPI_Recv (&a, 1, MPI_FLOAT, 0, 0, MPI_COMM_WORLD, &status);
   MPI_Send (&b, 1, MPI_FLOAT, 0, 0, MPI_COMM_WORLD);
   c = (a + b) / 2.0;
}
```

Before calling `MPI_Send`, process 0 blocks inside `MPI_Recv`, waiting for the message from process 1 to arrive. In the same way, process 1 blocks inside `MPI_Recv`, waiting for the message from process 0 to arrive. The processes are deadlocked.

Okay, that error was fairly obvious (though you might be surprised at how often this kind of bug occurs in practice). Let's consider a more subtle error that also leads to deadlock.

We're solving the same problem. Processes 0 and 1 wish to exchange floating-point values. Here is the code:

```
float      a, b, c;
int        id;                 /* Process rank */
MPI_Status status;
...
if (id == 0) {
   MPI_Send (&a, 1, MPI_FLOAT, 1, 1, MPI_COMM_WORLD);
   MPI_Recv (&b, 1, MPI_FLOAT, 1, 1, MPI_COMM_WORLD, &status);
   c = (a + b) / 2.0;
} else if (id == 1) {
   MPI_Send (&b, 1, MPI_FLOAT, 0, 0, MPI_COMM_WORLD);
   MPI_Recv (&a, 1, MPI_FLOAT, 0, 0, MPI_COMM_WORLD, &status);
   c = (a + b) / 2.0;
}
```

Now both processes send the data before trying to receive the data, but they still deadlock. Can you see the mistake? Process 0 sends a message with tag 1 and tries to receive a message with tag 1. Meanwhile, process 1 sends a message with tag 0 and tries to receive a message with tag 0. Both processes will block inside `MPI_Recv`, because neither process will receive a message with the proper tag.

Another common error occurs when the sending process sends the message to the wrong destination process, or when the receiving process attempts to receive the message from the wrong source process.

6.6 DOCUMENTING THE PARALLEL PROGRAM

We can now proceed with our parallel implementation of Floyd's algorithm. Our parallel program appears in Figure 6.9.

We use a typedef and a macro to indicate the type of matrix we are manipulating. If we decided to modify our program to find shortest paths in double-precision floating-point, rather than integer, matrices, we would only have to change these two lines as shown here:

```
typedef double dtype;
#define MPI_TYPE MPI_DOUBLE
```

Function `main` is responsible for reading and printing the original distance matrix, calling the shortest path function, and printing transformed distance matrix. Note that it checks to ensure the matrix is square. If the number of rows does not equal the number of columns, the processes collectively call function

`terminate`, which prints the appropriate error message, shuts down MPI, and terminates program execution. The source code for function `terminate` appears in Appendix B.

Now let's look at the function that actually implements Floyd's algorithm. Function `compute_shortest_paths` has four parameters: the process rank, the number of processes, a pointer to the process's portion of the distance matrix, and the size of the matrix.

Recall that during each iteration `k` of the algorithm, row `k` must be made available to every process, in order to perform the computation

```
a[i][j] = MIN(a[i][j],a[i][k]+a[k][j]);
```

```
/*
 *   Floyd's all-pairs shortest-path algorithm
 */

#include <stdio.h>
#include <mpi.h>
#include "MyMPI.h"

typedef int dtype;
#define MPI_TYPE MPI_INT

int main (int argc, char *argv[]) {
   dtype** a;          /* Doubly-subscripted array */
   dtype*  storage;    /* Local portion of array elements */
   int     i, j, k;
   int     id;         /* Process rank */
   int     m;          /* Rows in matrix */
   int     n;          /* Columns in matrix */
   int     p;          /* Number of processes */

   void compute_shortest_paths (int, int, int**, int);

   MPI_Init (&argc, &argv);
   MPI_Comm_rank (MPI_COMM_WORLD, &id);
   MPI_Comm_size (MPI_COMM_WORLD, &p);

   read_row_striped_matrix (argv[1], (void *) &a,
      (void *) &storage, MPI_TYPE, &m, &n, MPI_COMM_WORLD);

   if (m != n) terminate (id, "Matrix must be square\n");

   print_row_striped_matrix ((void **) a, MPI_TYPE, m, n,
      MPI_COMM_WORLD);
   compute_shortest_paths (id, p, (dtype **) a, n);
   print_row_striped_matrix ((void **) a, MPI_TYPE, m, n,
      MPI_COMM_WORLD);
   MPI_Finalize();
}
```

Figure 6.9 MPI program implementing Floyd's algorithm.

```
void compute_shortest_paths (int id, int p, dtype **a, int n)
{
   int  i, j, k;
   int  offset;   /* Local index of broadcast row */
   int  root;     /* Process controlling row to be bcast */
   int* tmp;      /* Holds the broadcast row */

   tmp = (dtype *) malloc (n * sizeof(dtype));
   for (k = 0; k < n; k++) {
      root = BLOCK_OWNER(k,p,n);
      if (root == id) {
         offset = k - BLOCK_LOW(id,p,n);
         for (j = 0; j < n; j++)
            tmp[j] = a[offset][j];
      }
      MPI_Bcast (tmp, n, MPI_TYPE, root, MPI_COMM_WORLD);
      for (i = 0; i < BLOCK_SIZE(id,p,n); i++)
         for (j = 0; j < n; j++)
            a[i][j] = MIN(a[i][j],a[i][k]+tmp[j]);
   }
   free (tmp);
}
```

Figure 6.9 (contd.) MPI program implementing Floyd's algorithm.

Every process allocates an array of n integers, called `tmp`, that will be used to store row k.

As in the sequential algorithm, the parallel algorithm has n iterations. During each iteration, the processes determine which process controls row k. This process is the root of the broadcast tree. After the call to `MPI_Bcast`, each process has a copy of row k in its array `tmp`. Hence the assignment shown previously becomes

```
a[i][j] = MIN(a[i][j],a[i][k]+tmp[j]);
```

6.7 ANALYSIS AND BENCHMARKING

It's easy to see that the sequential version of Floyd's algorithm has time complexity $\Theta(n^3)$. Let's analyze the complexity of our parallel version of Floyd's algorithm.

The innermost loop, the one that updates a single row of A, is identical to the innermost loop in the sequential algorithm and has time complexity $\Theta(n)$. Given a rowwise block-striped decomposition of matrix A, each process executes at most $\lceil n/p \rceil$ iterations of the middle loop. Hence the complexity of the inner two loops is $\Theta(n^2/p)$.

Immediately before the middle loop is the broadcast step. Passing a single message of length n from one processor to another has time complexity $\Theta(n)$. Since broadcasting to p processors requires $\lceil \log p \rceil$ message-passing steps, the overall time complexity of broadcasting each iteration is $\Theta(n \log p)$.

For every iteration of the outermost loop the parallel algorithm must compute the new root processor, which takes constant time. The root processor copies the correct row of A to array `tmp`, which takes $\Theta(n)$ time. The outermost loop executes n times.

Hence the overall time complexity of the parallel algorithm is

$$\Theta\big(n(1+n+n\log p+n^2/p)\big) = \Theta(n^3/p + n^2\log p)$$

Now let's come up with a prediction for the execution time of our parallel program on a commodity cluster. The parallel program requires n broadcasts. Each broadcast has $\lceil \log p \rceil$ steps. Each step involves passing messages that are $4n$ bytes long. Hence the expected communication time of the parallel program is

$$n\lceil \log p \rceil(\lambda + 4n/\beta)$$

If χ is the average time needed to update a single cell, then the expected computation time of the parallel program is $n^2\lceil n/p \rceil \chi$.

Adding computation time to broadcast time gives us a simple expression for the expected execution time of the parallel algorithm:

$$n^2\lceil n/p \rceil \chi + n\lceil \log p \rceil(\lambda + 4n/\beta)$$

However, this expression will overestimate the parallel execution time, because it ignores the fact that there can be considerable overlap between computation and communication.

See Figure 6.10, which illustrates the first four iterations of Floyd's algorithm executing on four processes, each on its own processor. Assume $n \geq 16$, so process 0 is the root process for the first four iterations. During each broadcast step, process 0 sends messages to processes 2 and 1. After it has initiated these messages,

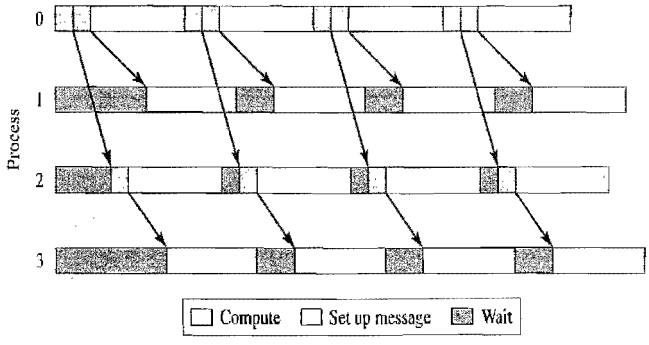


Figure 6.10 During the execution of the parallel version of Floyd's algorithm, there is significant overlap between message transmission (indicated by arrows) and computation.

it may begin updating its share of the rows of the matrix. Communications and computations overlap.

Examine process 1. It may not begin updating its portion of the matrix until it receives row 0 from process 0. During the first iteration, it must wait for the message to show up. However, this delay offsets its computational time frame from that of process 0. Process 1 completes its iteration 1 computation after process 0. Since process 0 initiates its transmission of the second row of the matrix to process 1 while process 1 is still working with the first row, process 1 will not have as long to wait for the second row.

In the figure, computation time per iteration exceeds the time needed to pass messages. For this reason, after the first iteration each process spends the same amount of time waiting for or setting up messages: $\lceil \log p \rceil \lambda$.

If $\lceil \log p \rceil 4n/\beta < \lceil n/p \rceil n\chi$, the message transmission time after the first iteration is completely overlapped by the computation time and should not be counted toward the total execution time. This is the case on our cluster when $n = 1000$. Hence a better expression for the expected execution time of the parallel program is

$$n^2\lceil n/p \rceil \chi + n\lceil \log p \rceil \lambda + \lceil \log p \rceil 4n/\beta$$

Figure 6.11 plots the predicted and actual execution times of our parallel program solving a problem of size 1000 on a commodity cluster, in which

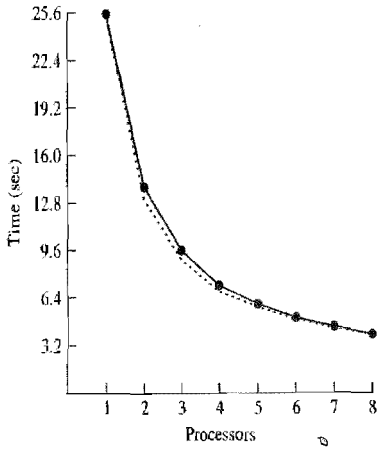


Figure 6.11 Predicted (dotted line) and actual (solid line) execution times of parallel implementation of Floyd's algorithm on a commodity cluster, solving a problem of size 1,000.

$\chi = 25.5$ nsec, $\lambda = 250$ μsec, and $\beta = 10^7$. The average error between the predicted and actual execution times on $2, \ldots, 7$ processors is 3.8 percent.

6.8 SUMMARY

We have developed a parallel version of Floyd's algorithm in C with MPI. The program achieves good speedup on a commodity cluster for moderately sized matrices. Our implementation uses point-to-point messages among pairs of processors. We have introduced the local communication functions `MPI_Send` and `MPI_Recv` that support point-to-point messages.

We have also begun the development of a library of functions that will eventually support the input, output, and redistribution of matrices and vectors with a variety of data decompositions. The two input/output functions referenced in this chapter are based on a rowwise block-striped decomposition of a matrix. Function `read_row_striped_matrix` reads a matrix from a file and distributes its elements to the processes in a group. Function `print_row_striped_matrix` prints the elements of a matrix distributed among a group of processes.

6.9 KEY TERMS

adjacency matrix
all-pairs shortest-path problem
directed graph
graph
point-to-point communication
weighted graph

6.10 BIBLIOGRAPHIC NOTES

Floyd's algorithm originally appeared in the *Communications of the ACM* in 1962 [27]. It is a generalization of Warshall's transitive closure algorithm, which appeared in the *Journal of the ACM* just a few months earlier [111].

Foster compares two parallel versions of Floyd's algorithm [31]. The first agglomerates primitive tasks in the same row, resulting in a rowwise block-striped data decomposition. The second agglomerates two-dimensional blocks of primitive tasks. In the next chapter we'll see this introduced as a "block checkerboard" decomposition. Foster shows that the second design is superior.

Grama et al. also describe a parallel implementation of Floyd's algorithm based on a block checkerboard data decomposition [44].

6.11 EXERCISES

6.1 Suppose we have chosen a block agglomeration of n elements (labeled $0, 1, \ldots, n-1$) to p processes (labeled $0, 1, \ldots, p-1$) in which

process i is responsible for elements $\lfloor in/p \rfloor$ through $\lfloor (i+1)n/p \rfloor - 1$. Prove that the last process is responsible for $\lceil n/p \rceil$ elements.

6.2 Reflect on the example of file input illustrated in Figure 6.6. What is the advantage of having process 3 input and pass along the data, rather than process 0?

6.3 Outline the changes that would need to be made to the parallel implementation of Floyd's all-pairs shortest-path algorithm if we decided to use a columnwise block-striped data distribution.

6.4 Outline the changes that would need to be made to the parallel implementation of Floyd's all-pairs shortest-path algorithm if we decided to use a rowwise interleaved striped decomposition (illustrated in Figure 12.3a).

6.5 Consider another version of Floyd's algorithm based on a third data decomposition of the matrix. Suppose p is a square number and n is a multiple of $\sqrt{p}$. In this data decomposition, each process is responsible for a square submatrix of A of size $(n/\sqrt{p}) \times (n/\sqrt{p})$.

a. Describe the communications necessary for every iteration of the outer loop of the algorithm.

b. Derive an expression for the communication time of the parallel algorithm, as a function of n, p, λ, and β.

c. Compare this communication time with the communication time of the parallel algorithm developed in this chapter.

6.6 Suppose the cluster used for benchmarking the parallel program developed in this chapter had 16 CPUs. Estimate the execution time that would result from solving a problem of size 1000 on 16 processors.

6.7 Assuming the same parallel computer used for the benchmarking in this chapter, estimate the execution time that would result from solving problems of size 500 and 2000 on $1, 2, \ldots, 8$ processors.

6.8 Assume that the time needed to send an n-byte message is $\lambda + n/\beta$. Write a program implementing the "ping pong" test to determine λ (latency) and β (bandwidth) on your parallel computer. Design the program to run on exactly two processes. Process 0 records the time and then sends a message to process 1. After process 1 receives the message, it immediately sends it back to process 0. Process 0 receives the message and records the time. The elapsed time divided by 2 is the average message-passing time. Try sending messages multiple times, and experiment with messages of different lengths, to generate enough data points that you can estimate λ and β.

6.9 Write your own version of `MPI_Reduce` using functions `MPI_Send` and `MPI_Recv`. You may assume that

datatype = `MPI_INT`,
operator = `MPI_SUM`, and
comm = `MPI_COMM_WORLD`

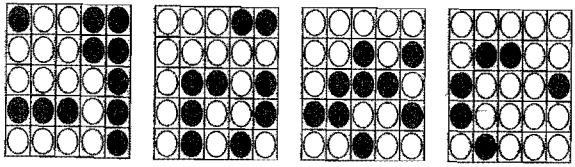


Figure 6.12 An initial state and three iterations of Conway's game of Life.

are updated simultaneously. Figure 6.12 illustrates three iterations of Life for a small grid of cells.

Write a parallel program that reads from a file an $m \times n$ matrix containing the initial state of the game. It should play the game of Life for j iterations, printing the state of the game once every k iterations, where j and k are command-line arguments.

CHAPTER

7

Performance Analysis

The highest and best form of efficiency is
the spontaneous cooperation of a free people.
Woodrow Wilson

7.1 INTRODUCTION

Being able to accurately predict the performance of a parallel algorithm you have designed can help you decide whether to actually go to the trouble of coding and debugging it. Being able to analyze the execution time exhibited by a parallel program can help you understand the barriers to higher performance and predict how much improvement can be realized by increasing the number of processors. This chapter will help you develop both of these skills.

We begin by deriving a general formula for the speedup achievable by a parallel program. We then look at well-known performance prediction formulas: Amdahl's Law, Gustafson-Barsis's Law, the Karp-Flatt metric, and the isoefficiency metric. Amdahl's Law can help you decide whether a program merits parallelization. Gustafson-Barsis's Law is a way to evaluate the performance of a parallel program. The Karp-Flatt metric can help you decide whether the principal barrier to speedup is the amount of inherently sequential code or parallel overhead. The isoefficiency metric is a way to evaluate the scalability of a parallel algorithm executing on a parallel computer. It can help you choose the design that will achieve higher performance when the number of processors increases.

7.2 SPEEDUP AND EFFICIENCY

We design and implement parallel programs in the hope that they will run faster than their sequential counterparts. **Speedup** is the ratio between sequential

execution time and parallel execution time:

$$\text{Speedup} = \frac{\text{Sequential execution time}}{\text{Parallel execution time}}$$

In the case studies we have worked through, we have discovered that the operations performed by parallel algorithm can be put into three categories:

- Computations that must be performed sequentially
- Computations that can be performed in parallel
- Parallel overhead (communication operations and redundant computations)

With these categories in mind, we can produce a simple model of speedup. Let $\psi(n, p)$ denote the speedup achieved solving a problem of size n on p processors, $\sigma(n)$ denote the inherently sequential (serial) portion of the computation, $\varphi(n)$ denote the portion of the computation that can be executed in parallel, and $\kappa(n, p)$ denote the time required for parallel overhead.

A sequential program, executing on a single processor, can only perform one computation at a time. Hence it requires time $\sigma(n) + \varphi(n)$ to execute the required computations. A sequential program requires no interprocessor communications, so the expression for sequential execution time does not have the $\kappa(n, p)$ term.

Now let's consider the best possible parallel execution time. The inherently sequential portion of the computation cannot benefit from parallelization. It contributes $\sigma(n)$ to the execution time of the parallel program, no matter how many processors are available. In the best case the portion of the computation that can be executed in parallel divides up perfectly among the p processors. In this case the time needed to perform these operations is $\varphi(n)/p$. Finally, we must add in time $\kappa(n, p)$ for the interprocessor communication required for the parallel program.

We have made the optimistic assumption that the parallel portion of the computation can be divided perfectly among the processors. If this is not the case, the parallel execution time will be larger, and the speedup will be smaller. Hence actual speedup will be less than or equal to the ratio between sequential execution time and parallel execution time as we have just defined. Here, then, is our completed expression for speedup:

$$\psi(n, p) \leq \frac{\sigma(n) + \varphi(n)}{\sigma(n) + \varphi(n)/p + \kappa(n, p)}$$

Adding processors reduces the computation time (by dividing the work among more processors) but increases the communication time. At some point the communication time increase is larger than the computation time decrease (see Figure 7.1). At this point the execution time begins to increase. Since speedup is inversely proportional to execution time, the speedup curve "elbows" and begins to decline.

The **efficiency** of a parallel program is a measure of processor utilization. We define efficiency to be speedup divided by the number of processors used:

$$\text{Efficiency} = \frac{\text{Sequential execution time}}{\text{Processors used} \times \text{Parallel execution time}}$$

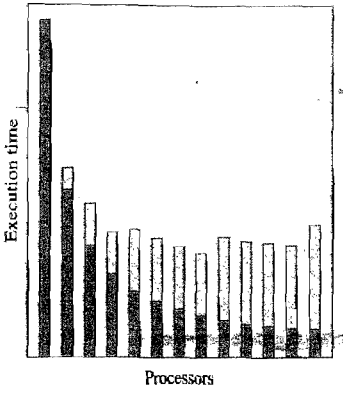


Figure 7.1 Nontrivial parallel algorithms have a computation component (black bars) that is a decreasing function of the number of processors used and a communication component (gray bars) that is an increasing function of the number of processors. For any fixed problem size there is an optimum number of processors that minimizes overall execution time.

More formally, let $\varepsilon(n, p)$ denote the efficiency of a parallel computation solving a problem of size n on p processors. Building on our earlier definition of speedup

$$\varepsilon(n, p) \leq \frac{\sigma(n) + \varphi(n)}{p(\sigma(n) + \varphi(n)/p + \kappa(n, p))}$$

$$\Rightarrow \varepsilon(n, p) \leq \frac{\sigma(n) + \varphi(n)}{p\sigma(n) + \varphi(n) + p\kappa(n, p)}$$

Since all terms are greater than or equal to zero, $0 \leq \varepsilon(n, p) \leq 1$.

7.3 AMDAHL'S LAW

Consider the expression for speedup we have just derived.

$$\psi(n, p) \leq \frac{\sigma(n) + \varphi(n)}{\sigma(n) + \varphi(n)/p + \kappa(n, p)}$$

Since $\kappa(n, p) > 0$,

$$\psi(n, p) \leq \frac{\sigma(n) + \varphi(n)}{\sigma(n) + \varphi(n)/p + \kappa(n, p)} \leq \frac{\sigma(n) + \varphi(n)}{\sigma(n) + \varphi(n)/p}$$

Let f denote the inherently sequential portion of the computation. In other words, $f = \sigma(n)/(\sigma(n) + \varphi(n))$. Then

$$\psi(n, p) \leq \frac{\sigma(n) + \varphi(n)}{\sigma(n) + \varphi(n)/p}$$

$$\Rightarrow \psi(n, p) \leq \frac{\sigma(n)/f}{\sigma(n) + \sigma(n)(1/f - 1)/p}$$

$$\Rightarrow \psi(n, p) \leq \frac{1/f}{1 + (1/f - 1)/p}$$

$$\Rightarrow \psi(n, p) \leq \frac{1}{f + (1 - f)/p}$$

Amdahl's Law [2]

Let f be the fraction of operations in a computation that must be performed sequentially, where $0 \leq f \leq 1$. The maximum speedup ψ achievable by a parallel computer with p processors performing the computation is

$$\psi \leq \frac{1}{f + (1 - f)/p}$$

Amdahl's Law is based on the assumption that we are trying to solve a problem of fixed size as quickly as possible. It provides an upper bound on the speedup achievable by applying a certain number of processors to solve the problem in parallel. It can also be used to determine the asymptotic speedup achievable as the number of processors increases.

EXAMPLE 1

Suppose we are trying to determine whether it is worthwhile to develop a parallel version of a program solving a particular problem. Benchmarking reveals that 90 percent of the execution time is spent inside functions that we believe we can execute in parallel. The remaining 10 percent of the execution time is spent in functions that must be executed on a single processor. What is the maximum speedup that we could expect from a parallel version of the program executing on eight processors?

■ Solution

By Amdahl's Law

$$\psi \leq \frac{1}{0.1 + (1 - 0.1)/8} \approx 4.7$$

We should expect a speedup of 4.7 or less.

EXAMPLE 2

If 25 percent of the operations in a parallel program must be performed sequentially, what is the maximum speedup achievable?

Solution

The maximum achievable speedup is

$$\lim_{p \to \infty} \frac{1}{0.25 + (1 - 0.25)/p} = 4$$

EXAMPLE 3

Suppose we have implemented a parallel version of a sequential program with time complexity $\Theta(n^2)$, where n is the size of the dataset. Assume the time needed to input the dataset and output the result is

$$(18000 + n)\ \mu\text{sec}$$

This constitutes the sequential portion of the program. The computational portion of the program can be executed in parallel; it has execution time

$$(n^2/100)\ \mu\text{sec}$$

What is the maximum speedup achievable by this parallel program on a problem of size 10,000?

Solution

By Amdahl's Law

$$\psi \leq \frac{(28{,}000 + 1{,}000{,}000)\ \mu\text{sec}}{(28{,}000 + 1{,}000{,}000/p)\ \mu\text{sec}}$$

The dashed line in Figure 7.2 is the upper bound on speedup derived from Amdahl's Law.

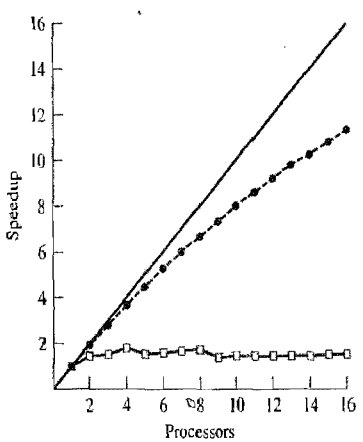


Figure 7.2 Speedup predicted by Amdahl's Law (dashed line) is higher than speedup prediction that takes communication overhead into account (solid line).

7.3.1 Limitations of Amdahl's Law

Amdahl's Law ignores overhead associated with the introduction of parallelism. Let's return to our previous example. Suppose the parallel version of the program has $\lceil \log n \rceil$ communication points. At each of these points, the communication time is

$$10{,}000\lceil \log p \rceil + (n/10)\ \mu\text{sec}$$

For a problem of size 10,000, the total communication time is

$$14(10{,}000\lceil \log p \rceil + 1{,}000)\ \mu\text{sec}$$

Now we have taken into account all of the factors included in our formula for speedup: $\sigma(n)$, $\varphi(n)$, and $\kappa(n, p)$. Our prediction for the speedup achievable by the parallel program solving a problem of size 10,000 on p processors is

$$\psi \leq \frac{(28{,}000 + 1{,}000{,}000)\ \mu\text{sec}}{(42{,}000 + 1{,}000{,}000/p + 140{,}000\lceil \log p \rceil)\ \mu\text{sec}}$$

The solid line in Figure 7.2 plots a new upper bound on speedup predicted by this more comprehensive formula. Taking communication time into account gives us a more realistic prediction of the parallel program's performance.

7.3.2 The Amdahl Effect

Typically, $\kappa(n, p)$ has lower complexity than $\varphi(n)$. That is the case with the hypothetical problem we have been considering: $\kappa(n, p) = \Theta(n \log n + n \log p)$, while $\varphi(n) = \Theta(n^2)$. Increasing the size of the problem increases the computation time faster than it increases the communication time. Hence for a fixed number of processors, speedup is usually an increasing function of the problem size. This is called the **Amdahl effect** [42]. Figure 7.3 illustrates the Amdahl effect by plotting expected speedup for our hypothetical problem. As problem size n increases, so does the height of the speedup curve.

7.4 GUSTAFSON-BARSIS'S LAW

Amdahl's Law assumes that minimizing execution time is the focus of parallel computing. It treats the problem size as a constant and demonstrates how increasing processors can reduce time.

Often, however, the goal of applying parallelism is to increase the accuracy of the solution that can be computed in a fixed amount of time. For example, an engineer studying airflow around the body of a hypersonic aircraft may want her computer to determine the solution to a problem in an hour (e.g., the length of a lunch break). If she has access to a computer with more processors, it is better for her to get a more detailed answer than to get the same results more quickly.

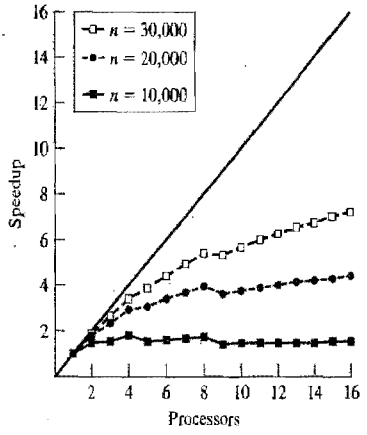


Figure 7.3 For any fixed number of processors, speedup is usually an increasing function of the problem size. This is called the **Amdahl effect.**

What happens if we treat time as a constant and let the problem size increase with the number of processors? The inherently sequential fraction of a computation typically decreases as problem size increases (the Amdahl effect). Increasing the number of processors enables us to increase the problem size, decreasing the inherently sequential fraction of a computation, and increasing the quotient between serial execution time and parallel execution time (speedup).

Consider the expression for speedup we have derived. Since $\kappa(n, p) \geq 0$,

$$\psi(n, p) \leq \frac{\sigma(n) + \varphi(n)}{\sigma(n) + \varphi(n)/p}$$

Let s denote the fraction of time spent in the *parallel* computation performing inherently sequential operations. The fraction of time spent in the parallel computation performing parallel operations is what remains, or $(1 - s)$. Mathematically,

$$s = \frac{\sigma(n)}{\sigma(n) + \varphi(n)/p}$$

$$(1 - s) = \frac{\varphi(n)/p}{\sigma(n) + \varphi(n)/p}$$

Hence

$$\sigma(n) = \big(\sigma(n) + \varphi(n)/p\big)s$$

$$\varphi(n) = \big(\sigma(n) + \varphi(n)/p\big)(1 - s)p$$

Therefore

$$\psi(n, p) \leq \frac{\sigma(n) + \varphi(n)}{\sigma(n) + \varphi(n)/p}$$

$$\Rightarrow \psi(n, p) \leq \frac{(\sigma(n) + \varphi(n)/p)(s + (1 - s)p)}{\sigma(n) + \varphi(n)/p}$$

$$\Rightarrow \psi(n, p) \leq s + (1 - s)p$$

$$\Rightarrow \psi(n, p) \leq p + (1 - p)s$$

Gustafson-Barsis's Law [46]
Given a parallel program solving a problem of size n using p processors, let s denote the fraction of total execution time spent in serial code. The maximum speedup ψ achievable by this program is

$$\psi \leq p + (1 - p)s$$

While Amdahl's Law determines speedup by taking a serial computation and predicting how quickly that computation could execute on multiple processors, Gustafson-Barsis's Law does just the opposite. It begins with a parallel computation and estimates how much faster the parallel computation is than the same computation executing on a single processor.

In many cases, assuming a single processor is only p times slower than p processors is overly optimistic. For example, imagine solving a problem on a parallel computer with 16 processors, each with one gigabyte of local memory. Suppose the dataset occupies 15 gigabytes, and the aggregate memory of the parallel computer is barely large enough to hold the dataset and multiple copies of the program. If we tried to solve the same problem on a single processor, the entire dataset would not fit in primary memory. If the working set of the executing program exceeded one gigabyte, it would begin to thrash, taking much more than 16 times as long to execute the parallel portion of the program as the group of 16 processors.

That is why we say that in Gustafson-Barsis's Law, speedup is the time required by a parallel computation divided into the time that *would* be required to solve the same problem on a single CPU, *if* it had sufficient memory. We refer to the speedup predicted by Gustafson-Barsis's Law as **scaled speedup**, because by using the parallel computation as the starting point, rather than the sequential computation, it allows the problem size to be an increasing function of the number of processors.

EXAMPLE 1

An application executing on 64 processors requires 220 seconds to run. Benchmarking reveals that 5 percent of the time is spent executing serial portions of the computation on a single processor. What is the scaled speedup of the application?

■ **Solution**

Since $s = 0.05$, the scaled speedup on 64 processors is

$$\psi = 64 + (1 - 64)(0.05) = 64 - 3.15 = 60.85$$

EXAMPLE 2

Vicki plans to justify her purchase of a $30 million Gadzooks supercomputer by demonstrating its 16,384 processors can achieve a scaled speedup of 15,000 on a problem of great importance to her employer. What is the maximum fraction of the parallel execution time that can be devoted to inherently sequential operations if her application is to achieve this goal?

■ **Solution**

Using Gustafson-Barsis's Law:

$$\begin{aligned} 15{,}000 &= 16{,}384 - 16{,}383s \\ \Rightarrow \quad s &= 1{,}384/16{,}383 \\ \Rightarrow \quad s &= 0.084 \end{aligned}$$

7.5 THE KARP-FLATT METRIC

Because Amdahl's Law and Gustafson-Barsis's Law ignore $\kappa(n, p)$, the parallel overhead term, they can overestimate speedup or scaled speedup. Karp and Flatt have proposed another metric, called the experimentally determined serial fraction, which can provide valuable performance insights [59].

Recall that we have represented the execution time of a parallel program executing on p processors as

$$T(n, p) = \sigma(n) + \varphi(n)/p + \kappa(n, p)$$

where $\sigma(n)$ is the inherently serial component of the computation, $\varphi(n)$ is the portion of the computation that may be executed in parallel, and $\kappa(n, p)$ is overhead resulting from processor communication and synchronization, and redundant computations. The serial program does not have any interprocessor communication or synchronization overhead, so its execution time is

$$T(n, 1) = \sigma(n) + \varphi(n)$$

We define the **experimentally determined serial fraction** e of the parallel computation to be

$$e = \big(\sigma(n) + \kappa(n, p)\big)/T(n, 1)$$

Hence

$$\sigma(n) + \kappa(n, p) = T(n, 1)e$$

We may now rewrite the parallel execution time as

$$T(n, p) = T(n, 1)e + T(n, 1)(1 - e)/p$$

Let's use ψ as a shorthand for $\psi(n, p)$. Since speedup $\psi = T(n, 1)/T(n, p)$, we have $T(n, 1) = T(n, p)\psi$. Hence

$$\begin{aligned}
& T(n, p) = T(n, p)\psi e + T(n, p)\psi(1 - e)/p \\
\Rightarrow \quad & 1 = \psi e + \psi(1 - e)/p \\
\Rightarrow \quad & 1/\psi = e + (1 - e)/p \\
\Rightarrow \quad & 1/\psi = e + 1/p - e/p \\
\Rightarrow \quad & 1/\psi = e(1 - 1/p) + 1/p \\
\Rightarrow \quad & e = \frac{1/\psi - 1/p}{1 - 1/p}
\end{aligned}$$

The Karp-Flatt Metric [59]

Given a parallel computation exhibiting speedup ψ on p processors, where $p > 1$, the experimentally determined serial fraction e is defined to be

$$e = \frac{1/\psi - 1/p}{1 - 1/p}$$

The experimentally determined serial fraction is a useful metric for two reasons. First, it takes into account parallel overhead [the $\kappa(n, p)$ term] that Amdahl's Law and Gustafson-Barsis's Law ignore. Second, it can help us detect other sources of overhead or inefficiency that are ignored in our simple model of parallel execution time. For example, we assume that p processors execute the parallelizable portion of the computation p times as quickly as a single processor. That is why the $\varphi(n)$ term in $T(n, 1)$ becomes the $\varphi(n)/p$ term in $T(n, p)$. This assumption ignores the fact that the amount of work to be done may not divide evenly among the processors. For example, suppose we have 19 equal and undividable pieces of work, each of which takes one unit of time to complete. If six processors are available, one processor must take four pieces while the other processors take three. The parallel execution time is 4, not 19/6.

For a problem of fixed size, the efficiency of a parallel computation typically decreases as the number of processors increases. By using the experimentally determined serial fraction, we can determine whether this efficiency decrease is due to (1) limited opportunities for parallelism or (2) increases in algorithmic or architectural overhead.

EXAMPLE 1

Benchmarking a parallel program on 1, 2, . . . , 8 processors produces the following speedup results:

p	2	3	4	5	6	7	8
ψ	1.82	2.50	3.08	3.57	4.00	4.38	4.71

What is the primary reason for the parallel program achieving a speedup of only 4.71 on eight processors?

■ Solution

Using the formula we have developed, we can compute the experimentally determined serial fraction e corresponding to each data point:

p	2	3	4	5	6	7	8
ψ	1.82	2.50	3.08	3.57	4.00	4.38	4.71
e	0.10	0.10	0.10	0.10	0.10	0.10	0.10

Since the experimentally determined serial fraction is not increasing with the number of processors, the primary reason for the poor speedup is the limited opportunity for parallelism—that is, the large fraction of the computation that is inherently sequential.

EXAMPLE 2

Benchmarking a parallel program on 1, 2, . . . , 8 processors produces the following speedup results:

p	2	3	4	5	6	7	8
ψ	1.87	2.61	3.23	3.73	4.14	4.46	4.71

What is the primary reason for the parallel program achieving a speedup of only 4.71 on eight processors?

■ Solution

We begin by computing the experimentally determined serial fraction for each of these program runs:

p	2	3	4	5	6	7	8
ψ	1.87	2.61	3.23	3.73	4.14	4.46	4.71
e	0.070	0.075	0.080	0.085	0.090	0.095	0.1

Since the experimentally determined serial fraction is steadily increasing as the number of processors increases, the principal reason for the poor speedup is parallel overhead. This could be time spent in process startup, communication, or synchronization, or it could be an architectural constraint.

7.6 THE ISOEFFICIENCY METRIC

Let's refer to a parallel program executing on a parallel computer as a **parallel system.** The **scalability** of a parallel system is a measure of its ability to increase performance as the number of processors increases.

As we have already seen, speedup (and hence efficiency) is typically an increasing function of the problem size, because the communication complexity is usually lower than the computational complexity. We call this the Amdahl effect. In order to maintain the same level of efficiency when processors are added, we can increase the problem size.

These ideas are formalized by the **isoefficiency relation.** To derive the isoefficiency relation, we return to our original definition of speedup:

$$\psi(n, p) \leq \frac{\sigma(n) + \varphi(n)}{\sigma(n) + \varphi(n)/p + \kappa(n, p)}$$

$$\Rightarrow \psi(n, p) \leq \frac{p\big(\sigma(n) + \varphi(n)\big)}{p\sigma(n) + \phi(n) + p\kappa(n, p)}$$

$$\Rightarrow \psi(n, p) \leq \frac{p\big(\sigma(n) + \varphi(n)\big)}{\sigma(n) + \phi(n) + (p-1)\sigma(n) + p\kappa(n, p)}$$

We define $T_o(n, p)$ to be the total amount of time spent by all processes doing work not done by the sequential algorithm. One component of this time is the time $p - 1$ processes spend executing inherently sequential code. The other component of this time is the time all p processes spend performing interprocessor communications and redundant computations. Hence $T_o(n, p) = (p-1)\sigma(n) + p\kappa(n, p)$. Substituting $T_o(n, p)$ into our previous equation, we get:

$$\Rightarrow \psi(n, p) \leq \frac{p\big(\sigma(n) + \varphi(n)\big)}{\sigma(n) + \phi(n) + T_o(n, p)}$$

Since efficiency equals speedup divided by p:

$$\varepsilon(n, p) \leq \frac{\sigma(n) + \varphi(n)}{\sigma(n) + \phi(n) + T_o(n, p)}$$

$$\Rightarrow \varepsilon(n, p) \leq \frac{1}{1 + \frac{T_o(n,p)}{\sigma(n)+\varphi(n)}}$$

Recalling that $T(n, 1)$ represents sequential execution time:

$$\Rightarrow \varepsilon(n, p) \leq \frac{1}{1 + T_o(n, p)/T(n, 1)}$$

$$\Rightarrow \frac{T_o(n, p)}{T(n, 1)} \leq \frac{1 - \varepsilon(n, p)}{\varepsilon(n, p)}$$

$$\Rightarrow T(n, 1) \geq \frac{\varepsilon(n, p)}{1 - \varepsilon(n, p)} T_o(n, p)$$

If we wish to maintain a constant level of efficiency, the fraction

$$\frac{\varepsilon(n, p)}{1 - \varepsilon(n, p)}$$

is a constant, and the formula simplifies to

$$T(n, 1) \geq CT_o(n, p)$$

Isoefficiency Relation [43]
Suppose a parallel system exhibits efficiency $\varepsilon(n, p)$. Define $C = \varepsilon(n, p)/(1 - \varepsilon(n, p))$ and $T_o(n, p) = (p - 1)\sigma(n) + p\kappa(n, p)$. In order to maintain the same level of efficiency as the number of processors increases, n must be increased so that the following inequality is satisfied:

$$T(n, 1) \geq CT_o(n, p)$$

We can use a parallel system's isoefficiency relation to determine the range of processors for which a particular level of efficiency can be maintained. Since parallel overhead increases when the number of processors increases, the way to maintain efficiency when increasing the number of processors is to increase the size of the problem being solved. The algorithms we are designing assume that the data structures we manipulate fit in primary memory. The maximum problem size we can solve is limited by the amount of primary memory that is available. For this reason we need to treat space as the limiting factor when we perform this analysis.

Suppose a parallel system has isoefficiency relation $n \geq f(p)$. If $M(n)$ denotes the amount of memory required to store a problem of size n, the relation $M^{-1}(n) \geq f(p)$ indicates how the amount of memory used must increase as a function of p in order to maintain a constant level of efficiency. We can rewrite this relation as $n \geq M(f(p))$. The total amount of memory available is a linear function of the number of processors used. Hence the function $M(f(p))/p$ indicates how the amount of memory used *per processor* must increase as a function of p in order to maintain the same level of efficiency. We call $M(f(p))/p$ the **scalability function.**

The complexity of $M(f(p))/p$ determines the range of processors for which a constant level of efficiency can be maintained, as illustrated in Figure 7.4. If $M(f(p))/p = \Theta(1)$, memory requirements per processor are constant, and the parallel system is perfectly scalable. If $M(f(p))/p = \Theta(p)$, memory requirements per processor increase linearly with the number of processors p. While memory is available, it is possible to maintain the same level of efficiency by increasing the problem size. However, since the memory used per processor increases linearly with p, at some point this value will reach the memory capacity of the system. Efficiency cannot be maintained when the number of processors increases beyond this point.

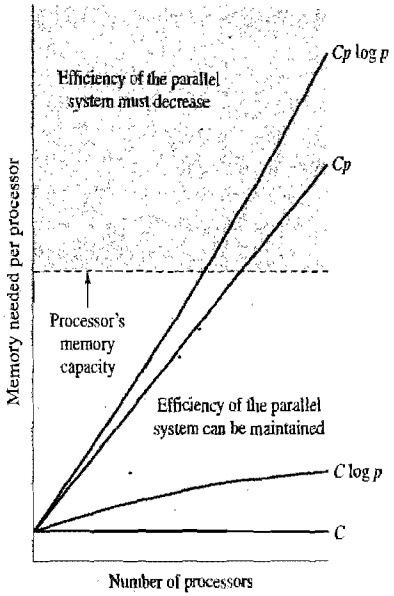


Figure 7.4 The way to maintain efficiency when increasing the number of processors is to increase the size of the problem being solved. The maximum problem size is limited by the amount of memory that is available, which is a linear function of the number of processors. Starting with the isoefficiency relation, and taking into account memory requirements as a function of n, we can determine how the amount of memory used per processor must increase as a function of p to maintain efficiency. The lower the complexity of this function, the more scalable the parallel system.

Similar arguments hold for the cases where $M(f(p))/p = \Theta(\log p)$ and $M(f(p))/p = \Theta(p \log p)$. While constants of proportionality must be taken into account, in general we say that the lower the complexity of $M(f(p))/p$, the higher the scalability of the parallel system.

EXAMPLE 1 Reduction

In Chapter 3 we developed a parallel reduction algorithm. The computational complexity of the sequential reduction algorithm is $\Theta(n)$. The reduction step has time complexity $\Theta(\log p)$. Every processor participates in this step, so $T_o(n, p) = \Theta(p \log p)$. Big-Oh notation ignores constants, but we can assume they are folded into the efficiency constant C.

Hence the isoefficiency relation for the reduction algorithm is

$$n \geq Cp \log p$$

The sequential algorithm reduces n values, so $M(n) = n$. Therefore,

$$M(Cp \log p)/p = Cp \log p/p = C \log p$$

We can mentally confirm that this result makes sense. Suppose we are sum-reducing n values on p processors. Each processor adds about n/p values, then participates in a reduction that has $\lceil \log p \rceil$ steps. If we double the number of processors and double the value of n, each processor's share is still about the same: n/p values. So the time each processor spends adding is the same. However, the number of steps needed to perform the reduction has increased slightly, from $\lceil \log p \rceil$ to $\lceil \log(2p) \rceil$. Hence the efficiency has dropped a bit. In order to maintain the same level of efficiency, we must more than double the value of n when we double the number of processors. The scalability function confirms this, when it shows that the problem size per processor must grow as $\Theta(\log p)$.

Floyd's Algorithm — EXAMPLE 2

Let's determine the isoefficiency function for the parallel implementation of Floyd's algorithm we developed in Chapter 6. The sequential algorithm has time complexity $\Theta(n^3)$. Each of the p processors executing the parallel algorithm spends $\Theta(n^2 \log p)$ time performing communications. Hence the isoefficiency relation is

$$n^3 \geq C(pn^2 \log p) \Rightarrow n \geq Cp \log p$$

This looks like the same relation we had in the previous example, but we have to be careful to consider the memory requirements associated with the problem size n. In the case of Floyd's algorithm the amount of storage needed to represent a problem of size n is n^2; that is, $M(n) = n^2$. The scalability function for this system is:

$$M(Cp \log p)/p = C^2 p^2 \log^2 p/p = C^2 p \log^2 p$$

This parallel system has poor scalability compared to parallel reduction.

Finite Difference Method — EXAMPLE 3

Consider a parallel algorithm implementing a finite difference method to solve a partial differential equation. (We'll consider these algorithms in more detail in Chapter 13.) The problem is represented by a $n \times n$ grid. Each processor is responsible for a subgrid of size $(n/\sqrt{p}) \times (n/\sqrt{p})$ (Figure 7.5). During each iteration of the algorithm every processor sends boundary values to its four neighbors; the time needed to perform these communications is $\Theta(n/\sqrt{p})$ per iteration.

The time complexity of the sequential algorithm solving this problem is $\Theta(n^2)$ per iteration.

The isoefficiency relation for this parallel system is

$$n^2 \geq Cp(n/\sqrt{p}) \Rightarrow n \geq C\sqrt{p}$$

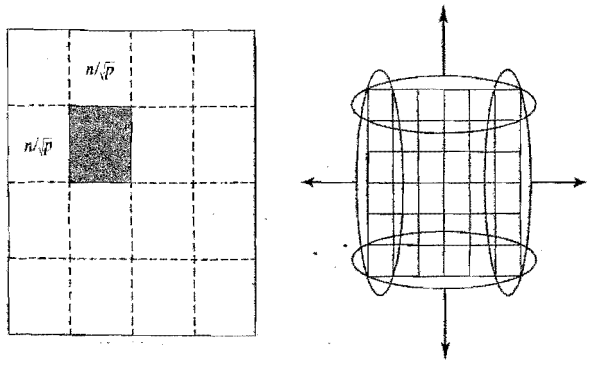


Figure 7.5 Partitioning for a parallel finite difference algorithm. (a) Each process is responsible for an $(n \times \sqrt{p}) \times (n \times \sqrt{p})$ block of the $n \times n$ matrix. (b) During each iteration each process sends $n/sqrt\ p$ boundary values to each of its four neighboring processes.

When we say a problem has size n, we mean the grid has n^2 elements. Hence $M(n) = n^2$ and

$$M(C\sqrt{p})/p = (C\sqrt{p})^2)/p = C^2p/p = C^2$$

The scalability function is $\Theta(1)$, meaning the parallel system is perfectly scalable.

7.7 SUMMARY

Our goal in parallel computing is to use p processors to execute a program p times faster than it executes on a single processor. The ratio of the sequential execution time to parallel execution time is called speedup.

$$\text{Speedup} = \frac{\text{Sequential execution time}}{\text{Parallel execution time}}$$

The efficiency of the parallel computation (also called processor utilization) is the speedup divided by the number of processors:

$$\text{Efficiency} = \frac{\text{Speedup}}{p}$$

To achieve a speedup of p, the parallel execution time must be $1/p$ that of the sequential program. Since there are only p processors, that means each processor must do an equal share of the work, all of the processors must be busy during the entire parallel execution, and there can be no extra operations introduced when the algorithm is made parallel. In other words, speedup equals p if and only if utilization equals 100 percent. In reality, this is rarely the case.

Why? First, there is usually some portion of the algorithm that cannot be performed on multiple processors. This is called serial code, and it prevents us from keeping all the processors busy all the time.

Second, virtually all parallel programs require at least some interactions among the processors. These communications operations do not exist in the sequential program. Hence they represent extra operations introduced when the algorithm is made parallel.

We have developed a general formula for speedup that takes into account the inherently sequential portion of the computation, the parallelizable portion of the computation, and parallel overhead (communication operations and redundant computations). We have also discussed four different lenses for analyzing the performance of parallel programs.

The first lens, Amdahl's Law, is forward looking. It relies upon an evaluation of the sequential program to predict an upper limit to the speedup that can be achieved by using multiple processors to speed the execution of the parallelizable portion of the program.

The second lens, Gustafson-Barsis's Law, is backward looking. It relies upon benchmarking of a parallel program to predict how long the program would take to run on a single processor, if that processor had enough memory. We say that Gustafson-Barsis's Law provides an estimate of scaled speedup, since the size of the problem is allowed to increase with the number of processors.

The third lens, the Karp-Flatt metric, examines the speedup achieved by a parallel program solving a problem of fixed size. The experimentally determined serial fraction can be used to support hypotheses about the performance of the program on larger numbers of processors.

The fourth and final lens, the isoefficiency metric, is used to determine the scalability of a parallel system. A parallel system is perfectly scalable if the same level of efficiency can be sustained as processors are added by increasing the size of the problem being solved. The scalability function, derived from the isoefficiency relation, identifies how the problem size must grow as a function of the number of processors in order to maintain the same level of efficiency.

7.8 KEY TERMS

Amdahl effect
Amdahl's Law
efficiency
experimentally determined serial fraction
Gustafson-Barsis's Law
isoefficiency relation
Karp-Flatt metric
parallel system
scalability
scalability function
scaled speedup
speedup

7.9 BIBLIOGRAPHIC NOTES

The seminal paper of Gustafson, Montry, and Benner [47] not only introduces the notion of scaled speedup, but also is the first to report scaled speedups in excess of 1000. It provides a fascinating glimpse into strategies for extracting maximum

speedup from a massively parallel computer (in their case a 1024-CPU nCUBE multicomputer). The authors won the Gordon Bell Award and the Karp Prize for this work.

You can find a much more detailed introduction to the isoefficiency metric in *Introduction to Parallel Computing* by Grama et al. [44]. Note that I have not adopted their definition of problem size. Grama et al. define problem size to be "the number of basic computation steps in the best sequential algorithm to solve the problem on a single processing element." In other words, when they say "problem size" they mean "sequential time." I believe this definition is counterintuitive, which is why I do not use it in this book.

7.10 EXERCISES

7.1 Using the definition of speedup presented in Section 7.2, prove that there exists a p_0 such that $p > p_0 \Rightarrow \psi(n, p) < \psi(n, p_0)$. Assume $\kappa(n, p) = C \log p$.

7.2 Starting with the definition of efficiency presented in Section 7.2, prove that $p' > p \Rightarrow \varepsilon(n, p') \leq \varepsilon(n, p)$.

7.3 Estimate the speedup achievable by the parallel reduction algorithm developed in Section 3.5 on $1, 2, \ldots, 16$ processors. Assume $n = 1{,}00{,}000$, $\chi = 10$ nanoseconds and $\lambda = 100$ μsec.

7.4 Benchmarking of a sequential program reveals that 95 percent of the execution time is spent inside functions that are amenable to parallelization. What is the maximum speedup we could expect from executing a parallel version of this program on 10 processors?

7.5 For a problem size of interest, 6 percent of the operations of a parallel program are inside I/O functions that are executed on a single processor. What is the minimum number of processors needed in order for the parallel program to exhibit a speedup of 10?

7.6 What is the maximum fraction of execution time that can be spent performing inherently sequential operations if a parallel application is to achieve a speedup of 50 over its sequential counterpart?

7.7 Shauna's parallel program achieves a speedup of 9 on 10 processors. What is the maximum fraction of the computation that may consist of inherently sequential operations?

7.8 Brandon's parallel program executes in 242 seconds on 16 processors. Through benchmarking he determines that 9 seconds is spent performing initializations and cleanup on one processor. During the remaining 233 seconds all 16 processors are active. What is the scaled speedup achieved by Brandon's program?

7.9 Courtney benchmarks one of her parallel programs executing on 40 processors. She discovers that it spends 99 percent of its time inside parallel code. What is the scaled speedup of her program?

7.10 The execution times of six parallel programs, labeled I-VI, have been benchmarked on $1, 2, \ldots, 8$ processors. The following table presents the speedups achieved by these programs.

	Speedup					
Processors	I	II	III	IV	V	VI
1	1.00	1.00	1.00	1.00	1.00	1.00
2	1.67	1.89	1.89	1.96	1.74	1.94
3	2.14	2.63	2.68	2.88	2.30	2.82
4	2.50	3.23	3.39	3.67	2.74	3.65
5	2.78	3.68	4.03	4.46	3.09	4.42
6	3.00	4.00	4.62	5.22	3.38	5.15
7	3.18	4.22	5.15	5.93	3.62	5.84
8	3.33	4.35	5.63	6.25	3.81	6.50

For each of these programs, choose the statement that *best* describes its likely performance on 16 processors:

A. The speedup achieved on 16 processors will probably be at least 40 percent higher than the speedup achieved on eight processors.

B. The speedup achieved on 16 processors will probably be less than 40 percent higher than the speedup achieved on eight processors, due to the large serial component of the computation.

C. The speedup achieved on 16 processors will probably be less than 40 percent higher than the speedup achieved on eight processors, due to the increase in overhead as processors are added.

7.11 Both Amdahl's Law and Gustafson-Barsis's Law are derived from the same general speedup formula. However, when increasing the number of processors p, the maximum speedup predicted by Amdahl's Law converges on $1/f$, while the speedup predicted by Gustafson-Barsis's Law increases without bound. Explain why this is so.

7.12 Given a problem to be solved and access to all the processors you care to use, can you always solve the problem within a specified time limit? Explain your answer.

7.13 Let $n \geq f(p)$ denote the isoefficiency relation of a parallel system and $M(n)$ denote the amount of memory required to store a problem of size n. Use the scalability function to rank the parallel systems shown below from most scalable to least scalable.

a. $f(p) = Cp$ and $M(n) = n^2$

b. $f(p) = C\sqrt{p}\log p$ and $M(n) = n^2$

c. $f(p) = C\sqrt{p}$ and $M(n) = n^2$

d. $f(p) = Cp\log p$ and $M(n) = n^2$

e. $f(p) = Cp$ and $M(n) = n$

f. $f(p) = p^C$ and $M(n) = n$. Assume $1 < C < 2$.

g. $f(p) = p^C$ and $M(n) = n$. Assume $C > 2$.

CHAPTER

Matrix-Vector Multiplication

Anarchy, anarchy! Show me a greater evil!
This is why cities tumble and the great houses rain down,
This is what scatters armies!

Sophocles, *Antigone*

8.1 INTRODUCTION

Matrix-vector multiplication is embedded in algorithms solving a wide variety of problems. For example, many iterative algorithms for solving systems of linear equations rely upon matrix-vector multiplication. The conjugate gradient method, which we will examine in Chapter 12, is such an algorithm.

Another practical use of matrix-vector multiplication is in the implementation of neural networks. Neural networks are used in such diverse applications as handwriting recognition, petroleum exploration, airline seating allocation, and credit card fraud detection [114]. The most straightforward way to determine the output values of a k-level neural network from its input values is to perform $k - 1$ matrix-vector multiplications. Moreover, training neural networks is typically done using the backpropagation algorithm, which also has matrix-vector multiplication at its core [98].

In this chapter we design, analyze, implement, and benchmark three MPI programs to multiply a dense square matrix by a vector. Each design is based upon a different distribution of the matrix and vector elements among the MPI processes. Altering the data decomposition changes the communication pattern among the processes, meaning different MPI functions are needed to route the data elements. Hence each of the three programs is significantly different from the other two.

In the course of developing these three programs we introduce four powerful MPI communication functions:

- `MPI_Allgatherv`, an all-gather function in which different processes may contribute different numbers of elements
- `MPI_Scatterv`, a scatter operation in which different processes may end up with different numbers of elements
- `MPI_Gatherv`, a gather operation in which the number of elements collected from different processes may vary
- `MPI_Alltoall`, an all-to-all exchange of data elements among processes

We also introduce five MPI functions that support grid-oriented communicators:

- `MPI_Dims_create`, which provides dimensions for a balanced Cartesian grid of processes
- `MPI_Cart_create`, which creates a communicator where the processes have a Cartesian topology
- `MPI_Cart_coords`, which returns the coordinates of a specified process within a Cartesian grid
- `MPI_Cart_rank`, which returns the rank of the process at specified coordinates in a Cartesian grid
- `MPI_Comm_split`, which partitions the processes of an existing communicator into one or more subgroups

8.2 SEQUENTIAL ALGORITHM

The sequential algorithm for multiplying a matrix by a vector appears in Figure 8.1. Matrix-vector multiplication is simply a series of inner product (or dot product) computations, as illustrated in Figure 8.2. Since computing an inner

Matrix-Vector Multiplication:

Input: $a[0..m-1, 0..n-1]$ — matrix with dimensions $m \times n$
$b[0..n-1]$ — vector with dimensions $n \times 1$

Output: $c[0..m-1]$ — vector with dimensions $m \times 1$

```
for i ← 0 to m − 1
  c[i] ← 0
  for j ← 0 to n − 1
    c[i] ← c[i] + a[i, j] × b[j]
  endfor
endfor
```

Figure 8.1 Sequential matrix-vector multiplication algorithm.

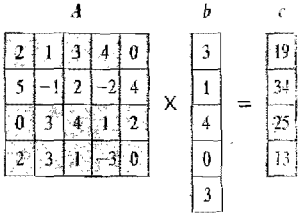


Figure 8.2 Matrix-vector multiplication can be viewed as a series of inner product (dot product) operations. For example, $c_1 = 5 \times 3 + (-1) \times 1 + 2 \times 4 + (-2) \times 0 + 4 \times 3 = 34$.

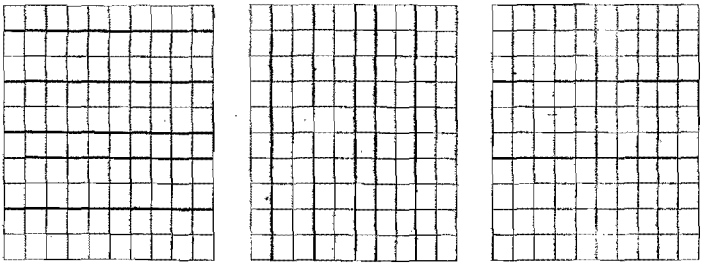


Figure 8.3 Three ways to decompose a two-dimensional matrix. In these examples a 10 × 10 matrix is decomposed among six processes. (a) Rowwise block-striped decomposition. (b) Columnwise block-striped decomposition. (c) Checkerboard block decomposition (processes are organized into a virtual 3 × 2 grid).

product of two n-element vectors requires n multiplications and $n - 1$ additions, it has complexity $\Theta(n)$. Matrix-vector multiplication performs m inner products; hence its complexity is $\Theta(mn)$. When the matrix is square, the algorithm's complexity is $\Theta(n^2)$.

8.3 DATA DECOMPOSITION OPTIONS

We use the domain decomposition strategy to develop our parallel algorithms. There are a variety of ways to partition, agglomerate, and map the matrix and vector elements. Each data decomposition results in a different parallel algorithm.

There are three straightforward ways to decompose an $m \times n$ matrix A: rowwise block striping, columnwise block striping, and checkerboard block decomposition (Figure 8.3). We have already seen the rowwise block-striped decomposition; it is how we divided the matrix elements among the processes

in our parallel implementation of Floyd's algorithm in Chapter 6. In this decomposition each of the p processes is responsible for a contiguous group of either $\lfloor m/p \rfloor$ or $\lceil m/p \rceil$ rows of the matrix.

A **columnwise block-striped decomposition** is analogous, except that the matrix is divided into groups of columns. Each of the p processes is responsible for a contiguous group of either $\lfloor n/p \rfloor$ or $\lceil n/p \rceil$ columns of the matrix.

In a **checkerboard block decomposition** the processes form a virtual grid, and the matrix is divided into two-dimensional blocks aligning with that grid. Assume the p processors form a grid with r rows and c columns. Each process is responsible for a block of the matrix containing at most $\lceil m/r \rceil$ rows and $\lceil n/c \rceil$ columns.

There are two natural ways to distribute vectors b and c. The vector elements may be **replicated**, meaning all the vector elements are copied on all of the tasks, or the vector elements may be divided among some or all of the tasks. In a **block decomposition** of an n-element vector, each of the p processes is responsible for a contiguous group of either $\lfloor n/p \rfloor$ or $\lceil n/p \rceil$ vector elements.

Why is it acceptable for a task to store vectors b and c in their entirety, but not matrix A? To simplify our argument, let's assume $m = n$. Vectors b and c contain only n elements, the same number of elements as in a single row or column of A. A task storing a row or column of A and single elements of b and c is responsible for $\Theta(n)$ elements. A task storing a row or column of A and all elements of b and c is responsible for $\Theta(n)$ elements. Hence whether the vectors are replicated or distributed in blocks among the tasks, the storage requirements are in the same complexity class.

With three ways to decompose the matrix and two ways to distribute the vector, six possible combinations result. In this chapter we consider three of the six combinations: a rowwise block-striped decomposition of the matrix and replicated vectors; a columnwise block-striped decomposition of the matrix and block-decomposed vectors; and a checkerboard block decomposition of the matrix and vectors block decomposed among the processes in the first column of the process grid.

8.4 ROWWISE BLOCK-STRIPED DECOMPOSITION

8.4.1 Design and Analysis

In this section we develop a parallel matrix-vector multiplication algorithm based on a domain decomposition that associates a primitive task with each row of the matrix A. Vectors b and c are replicated among the primitive tasks. A high-level view of the algorithm resulting from this domain decomposition appears in Figure 8.4. To compute an inner product, a primitive task needs a row and a column vector. Task i has row i of A and a copy of b, so it has all the data it needs to perform the inner product. After the inner product computation, task i

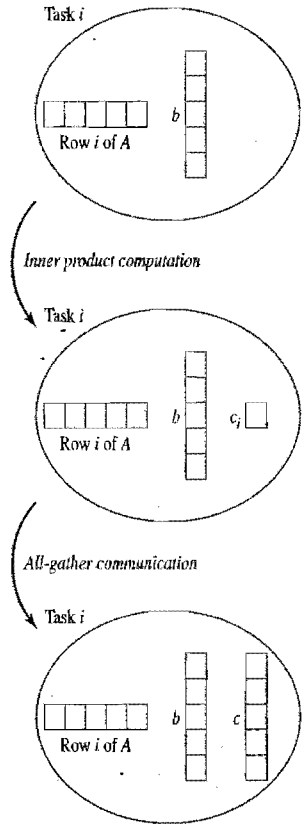


Figure 8.4 In our chosen domain decomposition, each primitive task has a row of the matrix and a copy of the vector. An inner product computation creates an element of the result vector c. An all-gather communication is needed to replicate vector c.

has element i of vector c. However, vectors are supposed to be replicated. An all-gather step communicates each task's element of c to all the other tasks, and the algorithm terminates.

In dense matrix-vector multiplication the number of computational steps needed to perform each inner product is identical. Hence our mapping strategy decision tree suggests we agglomerate primitive tasks associated with contiguous groups of rows and assign each of these combined tasks to a single process, creating a rowwise block-striped partitioning (shown in Figure 8.3a).

As we saw in Figure 8.4, at the end of the inner product computation, each primitive task computes a single element of the result vector. If the matrix decomposition is rowwise block striped, then each process (corresponding to a group of agglomerated tasks) will have a block of elements of the result vector.

When $m = n$, sequential matrix-vector multiplication has time complexity $\Theta(n^2)$. Let's determine the complexity of the parallel algorithm. Each process multiplies its portion of matrix A by vector b. No process is responsible for more than $\lceil n/p \rceil$ rows. Hence the complexity of the multiplication portion of the parallel algorithm is $\Theta(n^2/p)$.

In Chapter 3 we showed that in an efficient all-gather communication each process sends $\lceil \log p \rceil$ messages; the total number of elements passed is $n(p-1)/p$, when p is a power of 2. Hence the communication complexity of the parallel algorithm is $\Theta(\log p + n)$.

Combining the computational portion of the algorithm with the final all-gather communication step, the overall complexity of our parallel matrix-vector multiplication algorithm is $\Theta(n^2/p + n + \log p)$.

Now let's determine the isoefficiency of our parallel algorithm. The time complexity of the sequential algorithm is $\Theta(n^2)$. The only overhead in the parallel algorithm is performing the all-gather operation. When n is reasonably large, message transmission time in the all-gather operation is greater than the message latency. For this reason we simplify the communication complexity to $\Theta(n)$. Hence the isoefficiency function for the parallel matrix-vector multiplication algorithm based on a rowwise block-striped decomposition of the matrix is

$$n^2 \geq Cpn \Rightarrow n \geq Cp$$

When the problem size is n, the matrix has n^2 elements. Hence the memory utilization function $M(n) = n^2$. Computing the scalability function of our parallel algorithm:

$$M(Cp)/p = C^2p^2/p = C^2p$$

To maintain constant efficiency, memory utilization per processor must grow linearly with the number of processors. The algorithm is not highly scalable.

8.4.2 Replicating a Block-Mapped Vector

After each process performs its portion of the matrix-vector product, it has produced a block of result vector c. We must transform this block-mapped vector into a replicated vector, as shown in Figure 8.5.

Let's think about what we must to do to accomplish the transformation. First, each process needs to allocate memory to accommodate the entire vector, rather than just a piece of it. The amount of memory to be allocated depends on the type of the elements stored in the vector: characters, integers, floating-point numbers, or double-precision floating-point numbers, for example. Second, the processes must concatenate their pieces of the vector into a complete vector and

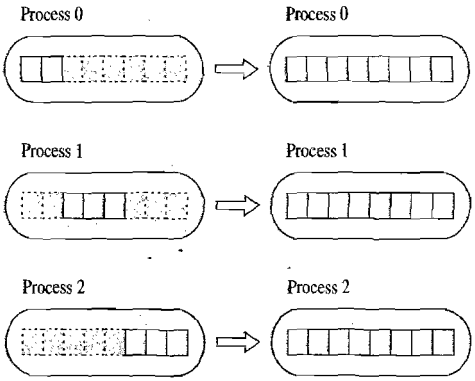


Figure 8.5 Transforming a block-distributed vector into a replicated vector. The elements of a block-distributed vector are distributed among the processes. Each element is stored exactly once. In contrast, when a vector is replicated, every process has every element.

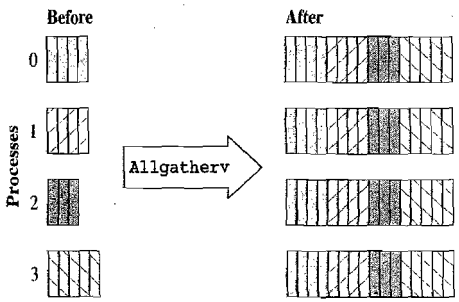


Figure 8.6 Function `MPI_Allgatherv` enables every process in a communicator to construct a concatenation of data items gathered from all of the processes in the communicator. If the same number of items is gathered from each process, the simpler function `MPI_Allgather` may be used.

share the results of the concatenation. Fortunately, a function that can perform the concatenation is in the MPI library.

8.4.3 Function `MPI_Allgatherv`

An **all-gather communication** concatenates blocks of a vector distributed among a group of processes and copies the resulting whole vector to all the processes. The correct MPI function is `MPI_Allgatherv` (illustrated in Figure 8.6).

If the same number of items is gathered from every process, the simpler function `MPI_Allgather` is appropriate. However, in a block decomposition of a vector, the number of elements per process is a constant only if the number of elements is a multiple of the number of processes. Since we cannot be assured of that, we will stick with `MPI_Allgatherv`.

Here is the function header:

```
int MPI_Allgatherv (void* send_buffer, int send_cnt,
   MPI_Datatype send_type, void* receive_buffer,
   int* receive_cnt, int* receive_disp,
   MPI_Datatype receive_type, MPI_Comm communicator)
```

Every parameter except the fourth is an input parameter:

`send_buffer`: the starting address of the data this process is contributing to the "all gather."

`send_cnt`: the number of data items this process is contributing.

`send_type`: the types of data items this process is contributing.

`receive_cnt`: an array indicating the number of data items to be received from each process (including itself).

`receive_disp`: an array indicating for each process the first index in the receive buffer where that process's items should be put.

`receive_type`: the type of the received elements.

`communicator`: the communicator in which this collective communication is occurring.

The fourth parameter, `receive_buffer`, is the address of the beginning of the buffer used to store the gathered elements.

Figure 8.7 illustrates function `MPI_Allgatherv` in action. Each process sets the scalar `send_cnt` to the number of elements in its block. Array `receive_cnt` contains the number of elements contributed by each process; it always has the same values on each process. In this example every process is concatenating elements in the same process order, so the values in array `receive_disp` are identical.

As we have seen, `MPI_Allgatherv` needs to be passed two arrays, each with one element per process. The first array indicates how *many* elements each process is contributing. The second array indicates the starting positions of these elements in the final, concatenated array.

We often encounter block mappings and in-order concatenation of array elements. We can write a function to build the two arrays needed for this common situation. Our function, called `create_mixed_xfer_arrays`, appears in Appendix B.

With these utilities in place, we can create a function to transform a block-distributed vector into a replicated vector, the last step in the matrix-vector multiplication algorithm. Function `replicate_block_vector` appears in Appendix B.

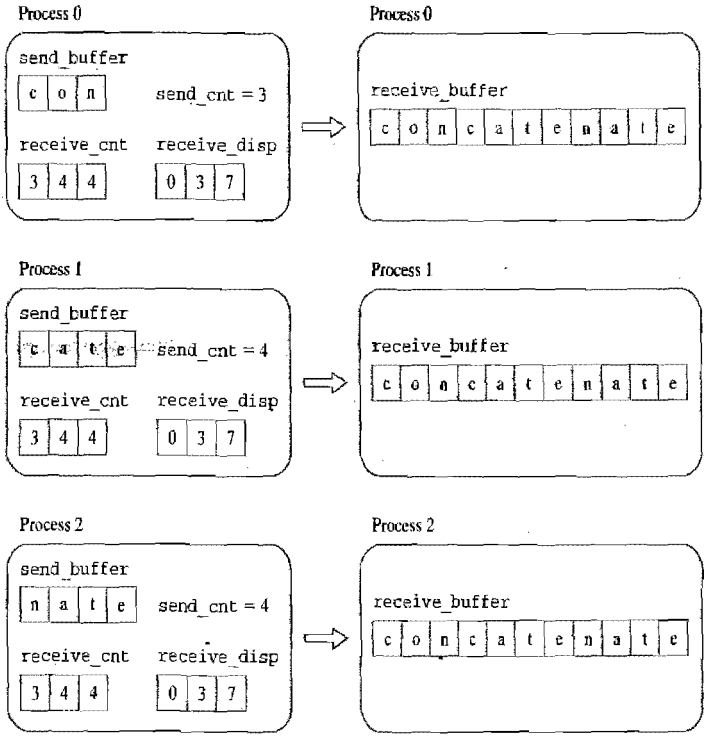


Figure 8.7 Example of how processes initialize variables `send_cnt`, `receive_cnt`, and `receive_disp` when performing a straightforward concatenation using function `MPI_Allgatherv`.

8.4.4 Replicated Vector Input/Output

We need a function to read a replicated vector from an input file. We are assuming the file was created with calls to `fwrite` and should be read with calls to `fread`. The file begins with an integer n representing the number of elements in the vector, followed by the n vector elements.

Process $p - 1$ tries to open the data file for reading. If it can open the file, it reads n and broadcasts it to the other processes. If it cannot open the file, it broadcasts a zero to the other processes. If that should happen, all the processes terminate execution. Otherwise, every process allocates memory to store the vector. Process $p - 1$ reads the vector and broadcasts it to the other processes. The source code for function `read_replicated_vector` appears in Appendix B.

From a parallel programming point of view, printing a replicated vector is simple. Typically we want a single process to do all the printing, to ensure that messages to standard output don't get scrambled. Since every process has a copy of the vector, all we have to do is ensure that only a single process

executes the calls to `printf`, and we're set. The source code for function `print_replicated_vector` is in Appendix B.

8.4.5 Documenting the Parallel Program

With the support functions in place, we can now write a parallel program to perform matrix-vector multiplication. Take another look at Figure 8.4, which summarizes the principal steps of our implementation. The complete C program appears in Figure 8.8.

We begin with the standard include files. We also include the header file `MyMPI.h` for the utility functions we have developed.

We want to be able to change the matrix and vector types with a minimum amount of program editing. In the body of the program we will use `dtype` to indicate the data type of the matrix and vector elements, and we will use `mpitype` as the type designator needed for MPI function calls. At the beginning of the program we use a typedef and a macro definition to establish values for `dtype` and `mpitype`.

After MPI initializations, the processes read and print matrix A. (We developed these functions in Chapter 6.) We also read and print vector b.

Each process allocates its portion of the result vector c and performs its share of the inner products.

At this point every process has a block of c. We convert c to a replicated vector, print it, and end program execution.

8.4.6 Benchmarking

Now let's develop an expression for the expected execution time of the parallel program on a commodity cluster. Let χ represent the time needed to compute a single iteration of the loop performing the inner product. We can determine χ by dividing the execution time of the sequential algorithm by n^2. The expected time for the computational portion of the parallel program is $\chi n \lceil n/p \rceil$.

The all-gather reduction requires each process to send $\lceil \log p \rceil$ messages. Each message has a latency λ. The total number of vector elements transmitted during the all-gather is $n(2^{\lceil \log p \rceil} - 1)/2^{\lceil \log p \rceil}$. Each vector element is a double-precision floating-point number occupying 8 bytes. Hence the expected execution time for the all-gather step is $\lambda \lceil \log p \rceil + 8n\left(2^{\lceil \log p \rceil} - 1\right)/2^{\lceil \log p \rceil}/\beta$.

Benchmarking on a commodity cluster of 450 MHz Pentium II processors connected by fast Ethernet reveals that $\chi = 63.4$ nsec, $\lambda = 250$ μsec, and $\beta = 10^6$ byte/sec.

Table 8.1 compares the actual and predicted execution times of our matrix-vector multiplication program solving a problem of size 1,000 on 1, 2, ..., 8 and 16 processors. The actual times reported in the table represent the average execution time over 100 runs of the parallel program. We determine the megaflops rate by dividing the total number of floating-point operations ($2n^2$) by the execution time, and then dividing by a million. The speedup of this program is illustrated in Figure 8.20 at the end of the chapter.

```
/*
 *   Matrix-vector multiplication, Version 1
 */

#include <stdio.h>
#include <mpi.h>
#include "MyMPI.h"

/* Change these two definitions when the matrix and vector
   element types change */

typedef double dtype;
#define mpitype MPI_DOUBLE

int main (int argc, char *argv[]) {
   dtype **a;          /* First factor, a matrix */
   dtype  *b;          /* Second factor, a vector */
   dtype  *c_block;    /* Partial product vector */
   dtype  *c;          /* Replicated product vector */
   dtype  *storage;    /* Matrix elements stored here */
   int     i, j;       /* Loop indices */
   int     id;         /* Process ID number */
   int     m;          /* Rows in matrix */
   int     n;          /* Columns in matrix */
   int     nprime;     /* Elements in vector */
   int     p;          /* Number of processes */
   int     rows;       /* Number of rows on this process */

   MPI_Init (&argc, &argv);
   MPI_Comm_rank (MPI_COMM_WORLD, &id);
   MPI_Comm_size (MPI_COMM_WORLD, &p);

   read_row_striped_matrix (argv[1], (void *) &a,
      (void *) &storage, mpitype, &m, &n, MPI_COMM_WORLD);
   rows = BLOCK_SIZE(id,p,m);
   print_row_striped_matrix ((void **) a, mpitype, m, n,
      MPI_COMM_WORLD);

   read_replicated_vector (argv[2], (void *) &b, mpitype,
      &nprime, MPI_COMM_WORLD);
   print_replicated_vector (b, mpitype, nprime,
      MPI_COMM_WORLD);

   c_block = (dtype *) malloc (rows * sizeof(dtype));
   c = (dtype *) malloc (n * sizeof(dtype));

   for (i = 0; i < rows; i++) {
      c_block[i] = 0.0;
      for (j = 0; j < n; j++)
         c_block[i] += a[i][j] * b[j];
   }

   replicate_block_vector (c_block, n, (void *) c, mpitype,
      MPI_COMM_WORLD);

   print_replicated_vector (c, mpitype, n, MPI_COMM_WORLD);
   MPI_Finalize();
   return 0;
}
```

Figure 8.8 Version 1 of parallel matrix-vector multiplication.

Table 8.1 Predicted versus actual performance of rowwise striped matrix-vector multiplication program multiplying a 1,000 x 1,000 matrix by a 1,000-element vector.

Processors	Predicted time	Actual time	Speedup	Megaflops
1	0.0634	0.0634	1.00	31.6
2	0.0324	0.0327	1.94	61.2
3	0.0223	0.0227	2.79	88.1
4	0.0170	0.0178	3.56	112.4
5	0.0141	0.0152	4.16	131.6
6	0.0120	0.0133	4.76	150.4
7	0.0105	0.0122	5.19	163.9
8	0.0094	0.0111	5.70	180.2
16	0.0057	0.0072	8.79	277.8

The parallel computer is a commodity cluster of 450 MHz Pentium IIs. Each processor has a fast Ethernet connection to a shared switch.

8.5 COLUMNWISE BLOCK-STRIPED DECOMPOSITION

8.5.1 Design and Analysis

In this section we will design another parallel matrix-vector multiplication algorithm, assuming that each primitive task i has column i of A and element i of vectors b and c. The structure of the resulting parallel algorithm is shown in Figure 8.9.

The computation begins with each task i multiplying its column of A by b_i, resulting in a vector of partial results. At the end of the computation task i needs only a single element of the result vector: c_i. What we need is an **all-to-all communication:** every partial result element j on task i must be transferred to task j. At this point every task i has the n partial results it needs to add in order to produce c_i.

Because every primitive task has identical computation and communication requirements, agglomerating them into larger tasks with the same number of columns (plus or minus one) ensures we have balanced the workload. Hence we will agglomerate the primitive tasks into p metatasks and map one metatask to each process.

In the previous section we assigned to each process a block of rows of A, which we called a rowwise block-striped decomposition. Now we will use a columnwise block-striped decomposition, agglomerating contiguous groups of columns of A, as shown in Figure 8.3b.

Let's determine the complexity of this parallel algorithm, assuming $m = n$. Each process multiplies its portion of matrix A by its block of vector b. No process is responsible for more than $\lceil n/p \rceil$ columns of A or elements of b. Hence the initial multiplication phase has time complexity $\Theta(n(n/p)) = \Theta(n^2/p)$. After the all-gather step, each processor sums the partial vectors collected from the other processors. There are p partial vectors, each of length at most $\lceil n/p \rceil$. The time

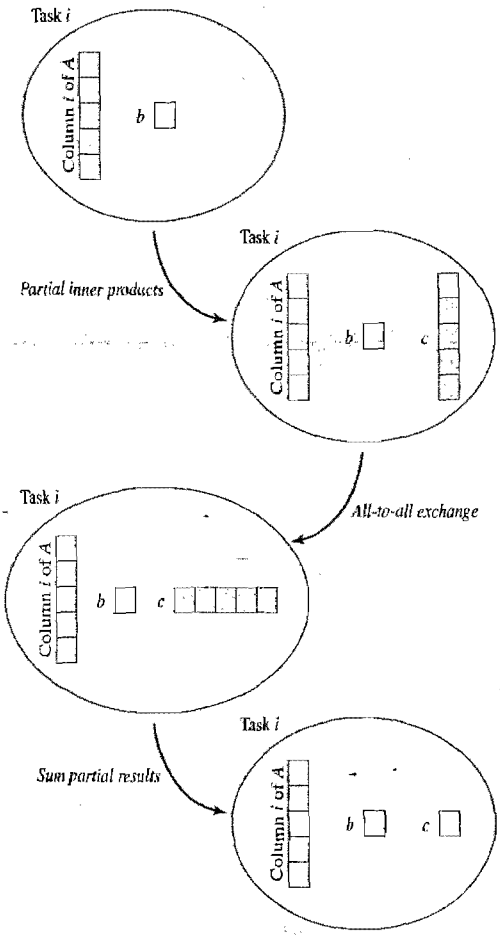


Figure 8.9 In this parallel matrix-vector multiplication algorithm each task has a column of the matrix and an element of the vector. An all-to-all communication moves the appropriate partial results to the tasks that will add them up.

complexity of this step is $\Theta(n)$. Therefore, the overall computational complexity of the parallel algorithm is $\Theta(n^2/p)$.

An all-to-all exchange can be performed in $\lceil \log p \rceil$ steps, using a hypercube communication pattern. During each step every process sends $n/2$ values to its partner and receives $n/2$ values from its partner. The total number of elements sent and received in the all-to-all exchange is $n\lceil \log p \rceil$. Hence the communication complexity of this implementation of all-gather is $\Theta(n \log p)$.

Another way to perform an all-to-all exchange is for each process to send a message to each of the other $p - 1$ processes. Every message contains just those

elements the destination process is supposed to receive from the source. In this implementation the total number of messages is $p - 1$, but the total number of elements passed by each process is less than or equal to n. The communication complexity of this algorithm is $\Theta(p + n)$.

When we combine the computational portion of the algorithm with the final all-gather communication step, the overall complexity of our parallel matrix-vector multiplication algorithm is either $\Theta(n^2/p + n \log p)$ or $\Theta(n^2/p + n + p)$, depending upon which way the all-to-all exchange is implemented.

Now let's determine the isoefficiency of the parallel algorithm. The time complexity of the sequential algorithm is $\Theta(n^2)$. The parallel overhead is limited to the all-to-all exchange operation. When n is reasonably large, the time for the all-to-all exchange is dominated by message transmission time rather than message latency. Using the second approach to implementing all-to-all exchange, we have $\Theta(n)$ complexity for this step, which is performed by all processes.

Hence the isoefficiency function for the parallel matrix-vector multiplication algorithm based on a columnwise block-striped decomposition of the matrix is

$$n^2 \geq Cpn \Rightarrow n \geq Cp$$

This is the same isoefficiency function we derived for the parallel algorithm based on a rowwise block-striped decomposition of the matrix. The parallel algorithm is not highly scalable, because in order to maintain a constant efficiency, memory used per processor must increase linearly with the number of processors.

8.5.2 Reading a Columnwise Block-Striped Matrix

Let's develop a function to read from a file a matrix stored in row-major order and distribute it among the processes in columnwise block-striped fashion. When a row-major matrix with multiple rows has a columnwise block-striped decomposition among multiple processes, the matrix elements controlled by a process are not stored as a contiguous group in the file. In fact, each row of the matrix must be scattered among all of the processes.

We will maintain our tradition of making a single process responsible for I/O. See Figure 8.10. In the first step, one process reads a row of the matrix into a temporary buffer. In step 2 that process scatters the elements of the buffer among all of the processes. The algorithm repeats these steps for the remaining rows of the matrix. The code for function `read_col_striped_matrix` appears in Appendix B.

Function `read_col_striped_matrix` makes use of MPI library routine `MPI_Scatterv` to distribute rows among the processes. Let's take a closer look at this function.

8.5.3 Function `MPI_Scatterv`

The MPI function `MPI_Scatterv` (Figure 8.11) enables a single root process to distribute a contiguous group of elements to all of the processes in a communicator, including itself.

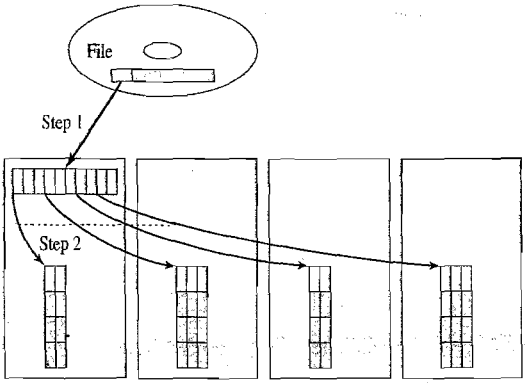


Figure 8.10 In a columnwise block-striped decomposition, each row of the matrix is distributed among the processors. One process inputs a row of the matrix (step 1) and then scatters its elements (step 2).

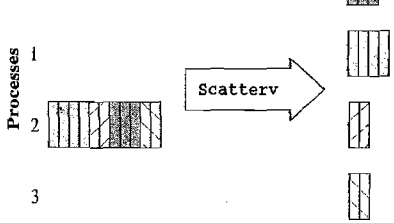


Figure 8.11 The collective communication function `MPI_Scatterv` allows a single MPI process to divide a contiguous group of data items and distribute unique portions to the rest of the processes in the communicator. If the same number of data items is distributed to every process, the simpler function `MPI_Scatter` is appropriate.

Here is the header of function `MPI_Scatterv`:

```
MPI_Scatterv (void *send_buffer, int* send_cnt,
   int* send_disp, MPI_Datatype send_type,
   void *recv_buffer, int recv_cnt,
   MPI_Datatype recv_type, int root, MPI_COMM communicator)
```

The function has nine parameters. All but the fifth are input parameters:

`send_buffer`: pointer to the buffer containing the elements to be scattered.

`send_cnt`: element i is the number of contiguous elements in `send_buffer` going to process i.

`send_disp`: element i is the offset in `send_buffer` of the first element going to process i.

`send_type`: the type of the elements in `send_buffer`.

`recv_buffer`: pointer to the buffer containing this process's portion of the elements received.

`recv_cnt`: the number of elements this process will receive.

`recv_type`: the type of the elements in `recv_buffer`.

`root`: the ID of the process with the data to scatter.

`communicator`: the communicator in which the scatter is occurring.

`MPI_Scatterv` is a collective communication function—all of the processes in a communicator participate in its execution. The function requires that each process has previously initialized two arrays: one that indicates the number of elements the root process should send to each of the other processes, and one that indicates the displacement of this block of elements in the array being scattered. In this case we want to scatter the blocks in process order: process 0 gets the first block, process 1 gets the second block, and so on. While we developed the function `create_mixed_xfer_arrays` in the context of a gather operation, we can use it in this context, too. The number of elements per process and the displacements are identical.

8.5.4 Printing a Columnwise Block-Striped Matrix

Now it's time to design a function to print a columnwise block-striped matrix. To ensure that values are printed in the correct order, we want only a single process to print all the values. In order to print a single row, a single process must gather together the elements of that row from the entire set of processes. Hence the data flow for this function is opposite that of function `read_col_striped_matrix`. The code for function `print_col_striped_matrix` appears in Appendix B.

Function `print_col_striped_matrix` makes use of MPI function `MPI_Gatherv` to collect row elements onto process 0, which then prints the row. The following subsection documents function `MPI_Gatherv`.

8.5.5 Function `MPI_Gatherv`

The MPI collective communication function `MPI_Gatherv` (Figure 8.12) performs this data-gathering function.

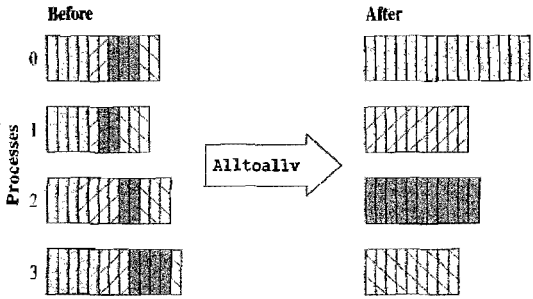


Figure 8.13 Function `MPI_Alltoallv` allows every MPI process to gather data items from all the processes in the communicator. The simpler function `MPI_Alltoall` should be used in the case where all of the groups of data items being transferred from one process to another have the same number of elements.

`send_type` is the type of the elements in `send_buffer`.

`recv_buffer` is the starting address of the buffer used to collect incoming elements (as well as the elements the process is sending to itself).

`recv_count` is an array; element i is the number of elements to receive from process i.

`recv_displacement` is an array; element i is the starting point in `recv_buffer` of the elements received from process i.

`recv_type` is the type the elements should be converted to before they are put in `recv_buffer`.

`communicator` indicates the set of processes participating in the all-to-all exchange.

8.5.8 Documenting the Parallel Program

We now have a firm foundation on which to build our second parallel matrix-vector multiplication program. The source for the program appears in Figure 8.14.

After the usual MPI initializations, we call function `read_col_striped_matrix` to input the contents of the data file containing a matrix and distribute it among the processes. We then print the matrix.

Similarly, we read vector b and print it.

Each process allocates memory to store `c_part_out`, the "outgoing" partial result vector. Most of these elements will end up on other processes. Each process also allocated memory for `c_part_in`, the "incoming" pieces of the other processes' partial result vectors.

Next is the actual computation. Each process multiplies its portion of the matrix (having dimensions `n` x `local_els`) by its portion of the vector (having length `local_else`), resulting in a partial result vector of length `n`.

```
/*
 *   Matrix-vector multiplication, Version 2
 */

#include <stdio.h>
#include <mpi.h>
#include "MyMPI.h"

/* Change these two definitions when the matrix and vector
   element types change */

typedef double dtype;
#define mpitype MPI_DOUBLE

int main (int argc, char *argv[]) {
   dtype **a;              /* The first factor, a matrix */
   dtype  *b;              /* The second factor, a vector */
   dtype  *c;              /* The product, a vector */
   dtype  *c_part_out;     /* Partial sums, sent */
   dtype  *c_part_in;      /* Partial sums, received */
   int    *cnt_out;        /* Elements sent to each proc */
   int    *cnt_in;         /* Elements received per proc */
   int    *disp_out;       /* Indices of sent elements */
   int    *disp_in;        /* Indices of received elements */
   int     i, j;           /* Loop indices */
   int     id;             /* Process ID number */
   int     local_els;      /* Cols of 'a' and elements of 'b'
                              held by this process */
   int     m;              /* Rows in the matrix */
   int     n;              /* Columns in the matrix */
   int     nprime;         /* Size of the vector */
   int     p;              /* Number of processes */
   dtype  *storage;        /* This process's portion of 'a' */

   MPI_Init (&argc, &argv);
   MPI_Comm_rank (MPI_COMM_WORLD, &id);
   MPI_Comm_size (MPI_COMM_WORLD, &p);

   read_col_striped_matrix (argv[1], (void ***) &a,
      (void **) &storage, mpitype, &m, &n, MPI_COMM_WORLD);
   print_col_striped_matrix ((void **) a, mpitype, m, n,
      MPI_COMM_WORLD);
   read_block_vector (argv[2], (void **) &b, mpitype,
      &nprime, MPI_COMM_WORLD);
   print_block_vector ((void *) b, mpitype, nprime,
      MPI_COMM_WORLD);

   /* Each process multiplies its columns of 'a' and vector
      'b', resulting in a partial sum of product 'c'. */

   c_part_out = (dtype *) my_malloc (id, n * sizeof(dtype));
   local_els = BLOCK_SIZE(id,p,n);

   for (i = 0; i < n; i++) {
      c_part_out[i] = 0.0;
```

Figure 8.14 Second parallel matrix-vector multiplication program.

```
            for (j = 0; j < local_els; j++)
               c_part_out[i] += a[i][j] * b[j];
         }

         create_mixed_xfer_arrays (id, p, n, &cnt_out, &disp_out);
         create_uniform_xfer_arrays (id, p, n, &cnt_in, &disp_in);
         c_part_in = (dtype*) my_malloc (id, p*local_els*sizeof(dtype));
         MPI_Alltoallv (c_part_out, cnt_out, disp_out, mpitype,
            c_part_in, cnt_in, disp_in, mpitype, MPI_COMM_WORLD);

         c = (dtype*) my_malloc (id, local_els * sizeof(dtype));
         for (i = 0; i < local_els; i++) {
            c[i] = 0.0;
            for (j = 0; j < p; j++)
               c[i] += c_part_in[i + j*local_els];
         }
         print_block_vector ((void *) c, mpitype, n, MPI_COMM_WORLD);
         MPI_Finalize();
         return 0;
      }
```

Figure 8.14 (contd.) Second parallel matrix-vector multiplication program.

The outgoing pieces of `c_part_out` have different sizes. A call to `create_mixed_xfer_arrays` sets up the counts and displacements for these pieces. In contrast, the incoming pieces of `c_part_in` all have the same size. Calling `create_uniform_xfer_arrays` correctly initializes the counts and displacements for these pieces. The MPI function `MPI_Alltoallv` performs the all-to-all communication, routing each piece to its destination.

Now each process has n chunks of length `local_els`. Adding these together yields its portion of the result vector `c`.

8.5.9 Benchmarking

Now let's develop an expression for the expected execution time of the parallel program on a commodity cluster. As before, χ is the time needed to compute a single iteration of the loop performing the inner product. The expected time for the computational portion of the parallel program is $\chi n\lceil n/p \rceil$.

The algorithm performs an all-to-all exchange of partially computed portions of the vector c. There are two common ways to perform an all-to-all exchange. The first way is for each process to send $\lceil \log p \rceil$ messages of length $n/2$. This requires that each process send $\lceil \log p \rceil$ messages and transmit a total of $\lceil \log p \rceil n/2$ data elements.

The second way is for each process to send directly to each of the other processes the elements destined for that process. This requires that each process send $p - 1$ messages and transmit a total of about $n(p-1)/p$ data elements.

For a large n, the message transmission time dominates the message latency, and the second approach is superior. Assuming each message has latency λ, the

Table 8.2 Comparison of predicted versus actual performance of our second matrix-vector multiplication program on a commodity cluster of 450 MHz Pentium IIs.

Processors	Predicted time	Actual time	Speedup	Megaflops
1	0.0634	0.0638	1.00	31.4
2	0.0324	0.0329	1.92	60.8
3	0.0222	0.0226	2.80	88.5
4	0.0172	0.0175	3.62	114.3
5	0.0143	0.0145	4.37	137.9
6	0.0125	0.0126	5.02	158.7
7	0.0113	0.0112	5.65	178.6
8	0.0104	0.0100	6.33	200.0
16	0.0085	0.0076	8.33	263.2

time needed to transmit a single byte is $1/\beta$, and the time needed to perform an all-gather of double-precision floating-point variables is $(p-1)(\lambda + 8n/(p\beta))$.

Benchmarking on a commodity cluster of 450 MHz Pentium II processors connected by fast Ethernet reveals that $\chi = 63.4$ nsec, $\lambda = 250$ μsec, and $\beta = 10^6$ byte/sec.

Table 8.2 compares the actual and predicted execution times of our matrix-vector multiplication program solving a problem of size 1,000 on $1, 2, \ldots, 8$ and 16 processors. The actual times reported in the table represent the average execution time over 100 runs of the parallel program. The speedup of this program is illustrated in Figure 8.20 at the end of the chapter.

8.6 CHECKERBOARD BLOCK DECOMPOSITION

8.6.1 Design and Analysis

In this domain decomposition we associate a primitive task with each element of the matrix. The task responsible for $a_{i,j}$ multiplies it by b_j, yielding $d_{i,j}$. Each element c_i of the result vector is $\sum_{j=0}^{n-1} d_{i,j}$. In other words, for each row i, we add all the $d_{i,j}$ terms to produce element i of vector c, as shown in Figure 8.15.

We agglomerate primitive tasks into rectangular blocks and associate a task with each block (as shown in Figure 8.3c). Since all the blocks have about the same size, the work required within each block is about the same, so we will set the block sizes so that we can map one task to each process. We can think of the processes as forming a two-dimensional grid. Vector b is distributed by blocks among the tasks in the first column of the task grid (Figure 8.16).

Now we can lay out the three principal steps of the parallel algorithm and the communications patterns necessary to accomplish these steps (see Figure 8.17). The task associated with matrix block $A_{i,j}$ performs a matrix-vector multiplication of this block with subvector b_j. Our first step, then, is to redistribute vector b so that each task has the correct portion of b. (We'll figure out how to do this

$$\begin{pmatrix} 4 & 5 & 3 \\ 6 & 2 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} \Rightarrow \begin{pmatrix} 4\times 1+5\times 2+3\times 3 \\ 6\times 1+2\times 2+1\times 3 \end{pmatrix} \Rightarrow \begin{pmatrix} 23 \\ 13 \end{pmatrix}$$

$$\begin{pmatrix} a_{0,0} & a_{0,1} & a_{0,2} \\ a_{1,0} & a_{1,1} & a_{1,2} \end{pmatrix} \begin{pmatrix} b_0 \\ b_1 \\ b_2 \end{pmatrix} \Rightarrow \begin{pmatrix} d_{0,0} & d_{0,1} & d_{0,2} \\ d_{1,0} & d_{1,1} & d_{1,2} \end{pmatrix} \Rightarrow \begin{pmatrix} c_0 \\ c_1 \end{pmatrix}$$

Figure 8.15 Our third parallel matrix-vector multiplication algorithm associates a primitive task with each element of matrix A. Every task multiplies its $a_{i,j}$ term with b_j, forming $d_{i,j}$. Rowwise reductions of the $d_{i,j}$ terms yield the elements of the product vector c.

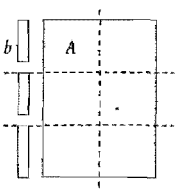


Figure 8.16 After agglomeration, tasks form a two-dimensional grid in which each task is responsible for a block of matrix A. This figure shows a 3×2 task grid. Vector b is divided into blocks allocated to tasks in the first column of the task grid.

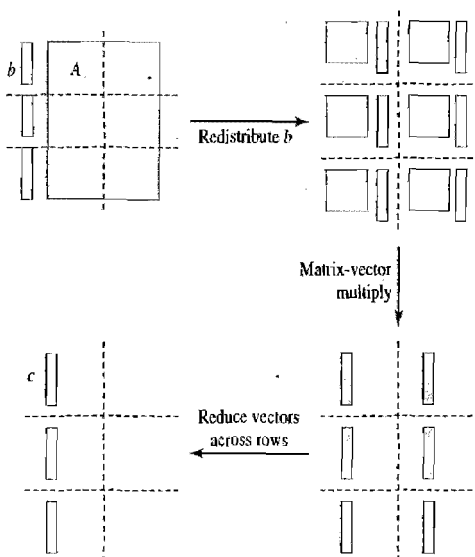


Figure 8.17 Phases of the parallel matrix-vector multiplication algorithm based on a checkerboard block decomposition of the matrix elements. First, vector b is distributed among the tasks. Second, each task performs matrix-vector multiplication on its block of matrix A and portion of vector b. Third, each row of tasks performs a sum-reduction of the result vectors, creating vector c.

a little later.) In the second step, each task performs a matrix-vector multiplication with its portions of A and b. In step 3, tasks in each row of the task grid perform a sum-reduction on their portions of c. After the sum-reduction, result vector c is distributed by blocks among the tasks in the first column of the task grid.

Now let's go back and figure out how to do the redistribution of vector b. Assume that the p tasks are divided into a $k \times l$ grid. Initially vector b is divided among the k tasks in the first column of the task grid. After the redistribution, a copy of b is divided among the l tasks in each row of the task grid.

If $k = l$, the redistribution is easier. See Figure 8.18a. The task at grid position $(i, 0)$ sends its portion of b to the task at grid position $(0, i)$. After this transfer each process in the first row of the task grid broadcasts its portion of b to the other tasks in the same column of the grid.

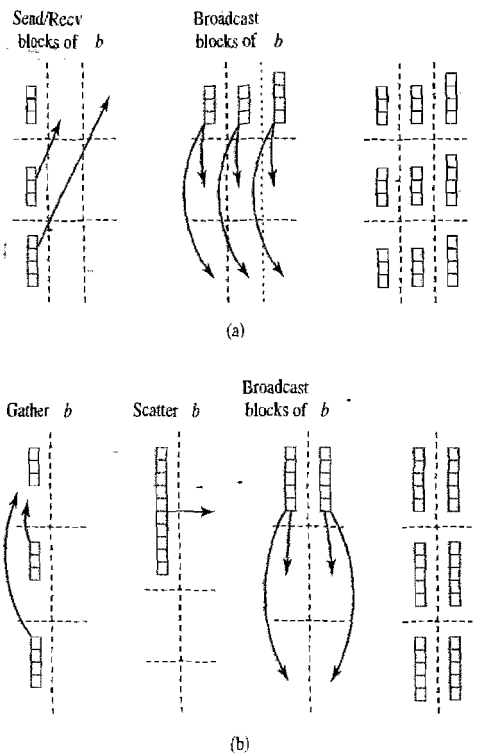


Figure 8.18 Redistributing vector b. (a) Algorithm is simpler when process grid is square. Processes in the first column send their blocks of b to processes in the first row. Then each process in the first row broadcasts its block of b to the other processes in its column. (b) An algorithm to handle the case when process grid is not square. First the processes in the first column gather vector b onto process at grid position (0, 0). Next the process at (0, 0) scatters b to the processes in the first row. Finally, each process in the first row broadcasts its block of b to the other processes in its column.

However, if $k \neq l$, the redistribution is more complicated, because the sizes of the blocks of b change. See Figure 8.18b. In this case, we first gather the elements of b onto the task at grid position $(0, 0)$. Next we scatter the elements of b among the tasks in the first row of the grid. Finally, each process in the first row of the task grid broadcasts its portion of b to the other tasks in the same column of the grid.

Let's analyze the complexity of the parallel algorithm, assuming $m = n$. We'll also assume that p is a square number and that the processes are arranged into a square grid. (Admittedly, this is the best-case assumption, but when the grid is size $p \times 1$, the decomposition devolves into a rowwise block-striped decomposition. Analogously, when the grid is size $1 \times p$, the decomposition devolves into a columnwise block-striped decomposition. We have already determined the complexity of parallel algorithms based on these two decompositions.)

Each process is responsible for a matrix block of size at most $\lceil n/\sqrt{p} \rceil \times \lceil n/\sqrt{p} \rceil$. Hence the time complexity of the matrix-vector multiplication step is $\Theta(n^2/p)$.

When p is a square number, the redistribution of b is done in two steps. First each process in the first column sends its portion of b to the process in the first row. The time required for this message-passing step is $\Theta(n/\sqrt{p})$. Next each process in the first row broadcasts its portion of b to the other processes in the same column. This broadcast has time complexity $\Theta(\log \sqrt{p}(n/\sqrt{p})) = \Theta(n \log p/\sqrt{p})$.

After the matrix-vector multiplication step, processes in each row sum-reduce their portions of b. The time needed for this communication operation is $\Theta(\log \sqrt{p}(n/\sqrt{p})) = \Theta(n \log p/\sqrt{p})$.

Combining these terms, the overall time complexity of the parallel matrix-vector multiplication algorithm based on a checkerboard block decomposition of the matrix is $\Theta(n^2/p + n \log p/\sqrt{p})$.

Now let's determine the isoefficiency of this parallel algorithm. The sequential algorithm has time complexity $\Theta(n^2)$. The parallel overhead is p times the communications complexity, or $np \log p/\sqrt{p} = n\sqrt{p} \log p$. Hence the isoefficiency function is

$$n^2 \geq Cn\sqrt{p} \log p \Rightarrow n \geq C\sqrt{p} \log p$$

Since $M(n) = n^2$, we get the following scalability function:

$$M(C\sqrt{p} \log p)/p = C^2 p \log^2 p/p = C^2 \log^2 p$$

This parallel algorithm is more scalable than the other two implementations of matrix-vector multiplication.

8.6.2 Creating a Communicator

Recall that a communicator is an opaque object that provides the environment for message-passing among processes. In the MPI programs we have implemented so far, the collective communications involved all of the processes, and we could rely upon the default communicator, `MPI_COMM_WORLD`. In our implementation

of matrix-vector multiplication based upon a checkerboard block decomposition of the matrix, there are four collective communication operations involving subsets of the processes:

- The processes in the first column of the virtual process grid participate in the communication that gathers vector b when p is not square.
- The processes in the first row of the virtual process grid participate in the communication that scatters vector b when p is not square.
- Each first-row process broadcasts its block of b to other processes in the same column of the virtual process grid.
- Each row of processes in the grid performs an independent sum-reduction, yielding vector c in the first column of processes.

In order to involve only a subset of the original process group in a collective communication operation, we need to create a new communicator.

A **communicator** consists of a **process group,** a **context,** and other properties called **attributes.** The topology of the processes is one of the most important attributes of a communicator. A **topology** allows you to associate an addressing scheme other than the rank with the processes. Topologies are virtual in the sense that they are not tied to the actual organization of the processors upon which the processes are executing. MPI supports two kinds of topologies: a Cartesian (or grid) topology and a graph topology. Our application requires the creation of a communicator with a Cartesian topology, a two-dimensional virtual grid of processes.

8.6.3 Function `MPI_Dims_create`

We want to create a virtual mesh of processes that is as close to square as possible, which results in an algorithm having maximum scalability. Passed the total number of nodes desired for a Cartesian grid and the number of grid dimensions, function `MPI_Dims_create` returns an array of integers specifying the number of nodes in each dimension of the grid, so that the sizes of the dimensions are as balanced as possible. The function has this header:

```
int MPI_Dims_create (int nodes, int dims, int *size)
```

The function has three parameters:

`nodes`: an input parameter, the number of processes in the grid.

`dims`: an input parameter, the number of dimensions in the desired grid.

`size`: an input/output parameter, the size of each grid dimension. The elements of `size` (`size[0],...,size[dims-1]`) must be initialized before calling the function. If `size[i]` = 0, the function is free to determine the size of that grid dimension. If `size[i]` > 0, the size of that dimension has been determined by the user.

For example, suppose we want to find the dimensions of a balanced two-dimensional grid containing p processes. The following code segment accomplishes this.

```
int p;
int size[2];
...
size[0] = size[1] = 0;
MPI_Dims_create (p, 2, size);
```

After function `MPI_Dims_create` has returned, `size[0]` contains the number of rows in the grid, and `size[1]` contains the number of columns in the grid.

8.6.4 Function `MPI_Cart_create`

After determining the size of each dimension of the virtual grid of processes, we want to create a communicator with this topology. Collective function `MPI_Cart_create` does this for us. It has this header:

```
int MPI_Cart_create (
   MPI_Comm old_comm, int dims, int *size, int *periodic,
   int reorder, MPI_Comm *cart_comm)
```

The function has five input parameters:

`old_comm`: the old communicator. All processes in the old communicator must collectively call the function.

`dims`: the number of grid dimensions.

`*size`: an array of size `dims`. Element `size[j]` is the number of processes in dimension `j`.

`*periodic`: an array of size `dims`. Element `periodic[j]` should be 1 if dimension `j` is periodic (communications wrap around the edges of the grid) and 0 otherwise.

`reorder`: a flag indicating if process ranks can be reordered. If `reorder` is 0, the rank of each process in the new communicator is the same as its rank in `old_comm`.

Function `MPI_Cart_create` has one output parameter. Through `cart_comm` it returns the address of the newly created Cartesian communicator.

Let's see how we would use this function in our application. The old communicator is `MPI_COMM_WORLD`. The grid has two dimensions. Function `MPI_Dims_create` initialized array `size` containing the size of each dimension. There are no wraparound communications, and we do not care if the ranks of the processes have the same order in the new communicator. These decisions lead to

the following code segment:

```
MPI_Comm cart_comm;   /* Cartesian topology communicator */
int p;                /* Processes */
int periodic[2];      /* Message wraparound flags */
int size[2];          /* Size of each grid dimension */
...
size[0] = size[1] = 0;
MPI_Dims_create (p, 2, size);
periodic[0] = periodic[1] = 0;
MPI_Cart_create (MPI_COMM_WORLD, 2, size, periodic,
                 1, &cart_comm);
```

8.6.5 Reading a Checkerboard Matrix

We will maintain our tradition of having a single process responsible for opening the data file containing the matrix, reading its contents, and distributing them to the appropriate processes. The distribution pattern is similar to the pattern we saw when we decomposed the matrix using columnwise striping. The difference is that instead of scattering each matrix row among all the processes, here we must scatter each row among a subset of the processes—those in the same row of the virtual process grid. See Figure 8.19. Process 0 is responsible for matrix input. Each time it reads in a row of the matrix, it sends the matrix row to the first process in the appropriate row of the process grid. After the receiving process reads the matrix row, it scatters it among the processes in its row of the process grid.

In order to accomplish all this, we need to add a trio of MPI functions to our repertoire.

8.6.6 Function `MPI_Cart_rank`

In order to send a matrix row to the first process in the appropriate row of the process grid, process 0 needs to know its rank. Function `MPI_Cart_rank`, when passed the coordinates of a process in the grid, returns its rank. It has this header:

```
MPI_Cart_rank (MPI_Comm comm, int *coords, int *rank)
```

The first parameter, `comm`, is an input parameter whose value is the Cartesian communicator in which the communication is occurring. The second parameter, `coords`, is another input parameter: an integer array containing the coordinates of a process in the virtual grid. The function returns through the third parameter, `rank`, the rank of the process in `comm` with the specified coordinates.

For example, suppose the virtual process grid has r rows. The matrix being read has m rows. Row i of the input matrix is mapped to the row of the process grid specified by `BLOCK_OWNER(i,r,m)`. The following code segment illustrates how process 0 could find the rank of the process that should receive the input row.

each of these new subgroups. It has this header:

```
int MPI_Comm_split (MIP_Comm old_comm, int partition,
                    int new_rank, MPI_Comm *new_comm)
```

The first three variables are input parameters:

`old_comm`: the existing communicator to which these processes belong. This is a collective function: every process in the old communicator must call it.

`partition`: the partition number.

`new_rank`: rank order of process within new communicator.

The function returns through `new_comm` a pointer to the new communicator to which this process belongs.

We've already seen how to use function `MPI_Cart_create` to create a Cartesian communicator that organizes the processes into a virtual two-dimensional grid, `grid_comm`. We've also seen how each process can call function `MPI_Cart_coords` to determine its coordinates in the grid, stored in the two-element array of integers `grid_coords`. Element `grid_coords[0]` is the row number, and element `grid_coords[1]` is the column number.

At this point we can use function `MPI_Comm_split` to partition the process grid into rows. Because we want to group together processes in the same row, we use the value of `grid_coords[0]` as the partitioning number. We rank processes according to their column indices by using `grid_coords[1]` as the determinant of the ranking order.

```
MPI_Comm grid_comm;        /* 2-D process grid */
MPI_Comm grid_coords[2];   /* Location of process in grid */
MPI_Comm row_comm;         /* Processes in same row */

MPI_Comm_split (grid_comm, grid_coords[0],
                grid_coords[1], &row_comm);
```

We will also use this function to divide the Cartesian communicator into separate communicators for every column in the process grid.

With the original grid communicator and these two additional communicators we can perform all the communications needed to redistribute the elements of b and perform the final sum-reduction resulting in c.

8.6.9 Benchmarking

We have written an MPI program implementing matrix-vector multiplication based on a checkerboard block decomposition of the matrix. Writing the key functions of this program is left as a series of exercises at the end of the chapter.

Let's develop an analytical model for the performance of this program on the same target cluster we used for the previous two programs. We will only consider the case where p is a square number.

Again, we let χ denote the time needed to perform a single iteration of the loop computing the inner product. Each process is responsible for a block of A having size at most $\lceil n/\sqrt{p} \rceil \times \lceil n/\sqrt{p} \rceil$. Hence the estimated computation time of the parallel program is $\chi \lceil n/\sqrt{p} \rceil \times \lceil n/\sqrt{p} \rceil$.

In the first step of the redistribution of b, the processes in the first column of the grid pass their blocks of b to processes in the first row of the grid. A process is responsible for at most $\lceil n/\sqrt{p} \rceil$ elements of b. Hence the time needed to send or receive a message containing these elements is $\lambda + 8\lceil n/\sqrt{p} \rceil/\beta$. In the second step of the redistribution, each process in the first row of the grid broadcasts its block of b to the other processes in the same column. The time required for this step is $\log \sqrt{p}(\lambda + 8\lceil n/\sqrt{p} \rceil/\beta)$.

After each process has performed its share of the matrix-vector multiplication, the processes in each row of the grid cooperate to reduce their portions of c. We ignore the time needed to perform the additions, which is swamped by the communications time. The communications time is the same as that needed to perform the columnwise broadcast: $\log \sqrt{p}(\lambda + 8\lceil n/\sqrt{p} \rceil/\beta)$.

Benchmarking on a commodity cluster of 450 MHz Pentium II processors connected by fast Ethernet reveals that $\chi = 63.4$ nanoseconds, $\lambda = 250$ μsec, and $\beta = 10^6$ byte/sec.

Table 8.3 compares the actual and predicted execution times of a checkerboard matrix-vector multiplication program solving a problem of size 1,000 on 1, 4, 9, and 16 processors. The actual times reported in the table represent the average execution time over 100 runs of the parallel program. The speedup of this program is compared with the speedups of our other two implementations of matrix-vector multiplication in Figure 8.20.

The number of messages sent by this program is virtually identical to the number of messages sent by the other two programs implementing matrix-vector multiplication. The principal difference between this algorithm and its predecessors is that the number of elements of b and c transmitted per process is $\Theta(n/\sqrt{p})$, whereas the other two algorithms transmitted $\Theta(n)$ elements. For this reason, we should expect the checkerboard algorithm to perform better than either the rowwise striped or the columnwise striped algorithms, once the number of processors gets large enough. Our experimental data bear this out. While the program based on the checkerboard decomposition does not outperform the other two programs on 1, 4, and 9 CPUs, it comes out on top when we increase the number of CPUs to 16.

Table 8.3 Predicted versus actual performance of checkerboard matrix-vector multiplication program multiplying a 1,000 x 1,000 matrix by a 1,000-element vector.

Processors	Predicted time	Actual time	Speedup	Megaflops
1	0.0634	0.0634	1.00	31.6
4	0.0178	0.0174	3.64	114.9
9	0.0097	0.0097	6.53	206.2
16	0.0062	0.0062	10.21	322.6

The parallel computer is a commodity cluster of 450 MHz Pentium IIs. Each processor has a fast Ethernet connection to a shared switch.

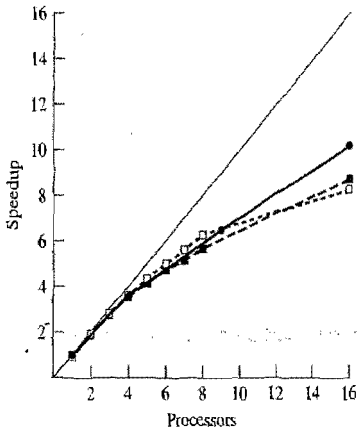


Figure 8.20 Speedup of three MPI programs multiplying a 1,000 × 1,000 matrix by a 1,000-element vector on a commodity cluster. The speedup of the program based on a rowwise block-striped decomposition of the matrix is indicated with a dotted line, the speedup of the columnwise block-striped implementation is shown with a dashed line, and the speedup of the checkerboard block implementation is drawn with a solid line.

8.7 SUMMARY

In this chapter we have designed, analyzed, and benchmarked three MPI programs to perform matrix-vector multiplication. The first design is based upon a rowwise block-striped decomposition of the matrix among the processes, the second springs from a columnwise block-striped decomposition, and the third arises from a checkerboard block decomposition of the matrix.

We have determined the isoefficiency of each of these algorithms. The checkerboard-decomposed algorithm has the best isoefficiency function, meaning it is more suitable for scaling to large numbers of processors than the algorithms based on the other two decompositions. Benchmarking reveals the superiority of the checkerboard-decomposed algorithm as the number of processors increases (Figure 8.20).

Because each algorithm is based on different matrix and vector decompositions, the resulting programs have much different communication patterns. As a result, we have encountered many powerful MPI communications functions for scattering and gathering data. We have also learned how to create communicators with a grid topology and how to partition the processes of a communicator into subgroups, each with its own communicator.

Compared to a C program performing matrix-vector multiplication, our parallel programs and their supporting functions are many times longer. The actual amount of code spent performing the computations is about the same, but reading and distributing matrices and vectors is much more complicated in a parallel setting. Developing and debugging these functions is a tedious process. That is why it makes sense to make them as general as possible, put them in a library, and reuse them. It also makes sense to use freely available libraries. The ScaLAPACK project resulted in the creation of a large suite of MPI-compatible functions supporting computational science and engineering. See the Bibliographic Notes section for more information on ScaLAPACK.

8.8 KEY TERMS

all-gather communication
all-to-all communication
attributes
block-decomposed vector
checkerboard block decomposition
columnwise block-striped decomposition
communicator
context
distributed vector
process group
replicated vector
topology

8.9 BIBLIOGRAPHIC NOTES

Other introductions to parallel matrix-vector multiplication algorithms include Pacheco [89] (rowwise striped), Bertsekas and Tsitsiklis [9] (rowwise striped and columnwise striped), Fox et al. [33] (checkerboard), and Grama et al. [44] (rowwise striped and checkerboard).

In the mid-1990s several U.S. government agencies provided funding to the ScaLAPACK project, a collaborative effort between Oak Ridge National Laboratory, Rice University, the University of California, Berkeley, the University of California, Los Angeles, the University of Illinois, and the University of Tennessee, Knoxville. These institutions developed many MPI-compatible libraries of numerical functions. These freely available libraries serve a wide variety of functions, including performing basic operations on matrices and vectors, solving linear systems of equations, computing eigenvalues and eigenvectors, and preconditioning matrices for iterative solvers. See `www.netlib.org/scalapack/` for more information about these libraries.

The March 1994 issue of *Communications of the ACM* focuses on artificial intelligence. It contains three survey articles on neural networks.

8.10 EXERCISES

8.1 Benchmark the parallel matrix-vector multiplication programs developed in this chapter on your parallel computer, this time including time spent reading the matrix and the vector from files. Which program exhibits higher performance? Why?

8.2 Using the performance model developed in Section 8.4 of this chapter, estimate the execution time, speedup, and megaflops rate of the first matrix-vector multiplication program on 9, 10, ..., 15 processors, assuming $n = 1{,}000$.

8.3 Write a matrix-vector multiplication program in which matrices are distributed among the processes in block-row fashion and vectors are distributed among the processes as blocks. You may assume both the matrix and the vector are input from a data file, using the same format as described for the example programs in this chapter. At the end of the program's execution, the result vector c should be distributed among the processes as blocks.

8.4 Implement another version of function `read_col_striped_matrix`. As in the function described in the chapter, a single process should be responsible for opening and reading the contents of the file. Unlike the function described in the chapter, the matrix distribution should be accomplished with $p - 1$ simple send-receive messages. For each row of the matrix, the process reading the file should call `MPI_Send` $p - 1$ times; each of the other processes should call `MPI_Recv` once. No process, including the process reading the file, should allocate memory for more than $n\lceil n/p \rceil$ matrix elements.

8.5 Implement another version of function `read_col_striped_matrix` that requires no calls to MPI functions. Each process opens the data file and reads its portion of the matrix.

8.6 Write a matrix-vector multiplication program in which matrices are distributed among the processes in block-column fashion and vectors are replicated. You may assume both the matrix and the vector are input from a data file, using the same format as described for the example programs in this chapter. At the end of the program's execution, vector c should be replicated.

8.7 Assume `grid_comm` is a communicator with a Cartesian topology that organizes processes into a two-dimensional grid. Write a code segment that partitions the process grid into columns. At the end of the code segment, each process's value of `col_comm` should be a communicator containing the calling process and all other processes in the same column of the process grid, but no others.

8.8 Suppose `grid_comm` is a communicator with a Cartesian topology that organizes processes into a virtual two-dimensional grid. Write a code segment illustrating how function `read_block_vector` can be used to open a file containing a vector and distribute it among the first column of processes in `grid_comm`. The name of the file is "Vector," and it contains double-precision floating-point values.

8.9 As part of a program that implements matrix-vector multiplication based on a checkerboard decomposition of the matrix, write a function that redistributes vector b. Assume the vector has n elements and the

processes are organized into a grid with dimensions $r \times c$. Initially vector b is distributed by blocks among the processes in the first column of the grid. The process at grid location $(i, 0)$ is responsible for `BLOCK_SIZE(i,r,n)` elements of b, beginning with the element having index `BLOCK_LOW(i,r,n)`. After the redistribution, every process in column j of the grid is responsible for `BLOCK_SIZE(j,c,n)` elements of b, beginning with the element having index `BLOCK_LOW(j,c,n)`.

a. Assume p is a square number; i.e., $r = c$.

b. Assume p is not a square number; i.e., $r \neq c$.

c. Make no assumptions about the value of p.

8.10 Write a program that implements matrix-vector multiplication based on a checkerboard block decomposition of the matrix. The program should read the matrix and the vector from an input file and print the answer to standard output. The names of the files containing the matrix and the vector should be specified as command-line arguments.

8.11 Write a function to transpose an $n \times n$ matrix A. Assume that before the function call, A is rowwise block decomposed among the p processes. After the function returns, A should be columnwise block decomposed among the p processes.

8.12 A binary search tree is a way of organizing n keys from a linearly ordered set to ensure their retrieval in $\Theta(\log n)$ time. If we know the probability of each key being accessed, we can create an optimal binary search tree that minimizes the average search time (Figure 8.21).

brown	0.16
dog	0.13
fox	0.06
jumped	0.08
lazy	0.07
over	0.17
quick	0.05
the	0.28

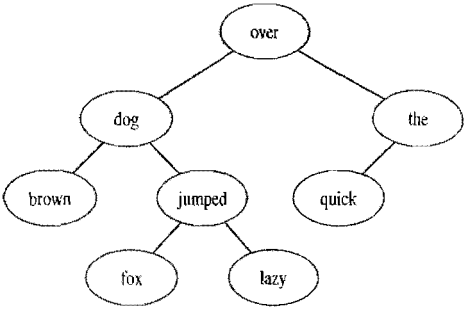


Figure 8.21 Given a set of keys and the probability of each key being accessed, an optimal binary search tree minimizes the average time needed to retrieve a key.

Figure 8.22 is a C program implementing a dynamic programming algorithm to find an optimal binary search tree, given a set of n probabilities. The program is adapted from pseudocode in Baase and Van Gelder [5]; consult their textbook for more details on the algorithm.

```
/*
 *  Given p[0], p[1], ..., p[N-1],the probability of each key
 *  in an ordered list of keys being the target of a search,
 *  this program uses dynamic programming to compute the
 *  optimal binary search tree that minimizes the average
 *  number of comparisons needed to find a key.
 *
 *  Last modification: 12 September 2002
 */

#include <stdio.h>
#include <values.h>

main (int argc, char *argv[]) {
   float bestcost;     /* Lowest cost subtree found so far */
   int   bestroot;     /* Root of lowest cost subtree */
   int   high;         /* Highest key in subtree */
   int   i, j;
   int   low;          /* Lowest key in subtree */
   int   n;            /* Number of keys */
   int   r;            /* Possible subtree root */
   float rcost;        /* Cost of subtree rooted by r */
   int   **root;       /* Best subtree roots */
   float **cost;       /* Best subtree costs */
   float *p;           /* Probability of each key */

   void alloc_matrix (void ***, int, int, int);
   void print_root (int **, int, int);

   /* Input the number of keys and probabilities */

   scanf ("%d", &n);
   p = (float *) malloc (n * sizeof(float));
   for (i = 0; i < n; i++)
      scanf ("%f", &p[i]);

   /* Find optimal binary search tree */

   alloc_matrix ((void ***) &cost, n+1, n+1, sizeof(float));
   alloc_matrix ((void ***) &root, n+1, n+1, sizeof(int));
   for (low = n; low >= 0; low--) {
      cost[low][low] = 0.0;
      root[low][low] = low;
      for (high = low+1; high <= n; high++) {
         bestcost = MAXFLOAT;
         for (r = low; r < high; r++) {
            rcost = cost[low][r] + cost[r+1][high];
            for (j = low; j < high; j++) rcost += p[j];
            if (rcost < bestcost) {
               bestcost = rcost;
               bestroot = r;
            }
         }
```

Figure 8.22 C program implementing dynamic programming algorithm to find an optimal binary search tree.

```
            cost[low][high] = bestcost;
            root[low][high] = bestroot;
         }
      }

      /* Print structure of binary search tree */

      print_root(root, 0, n-1);
}

/* Print the root of the subtree spanning keys
   'low' through 'high' */

void print_root (int **root, int low, int high) {
   printf ("Root of tree spanning %d-%d is %d\n",
      low, high, root[low][high+1]);
   if (low < root[low][high+1]-1)
      print_root (root, low, root[low][high+1]-1);
   if (root[low][high+1] < high-1)
      print_root (root, root[low][high+1]+1, high);
}

/* Allocate a two-dimensional array with 'm' rows and
   'n' columns, where each entry occupies 'size' bytes */

void alloc_matrix (void ***a, int m, int n, int size)
{
   int i;
   void *storage;
   storage = (void *) malloc (m * n * size);
   *a = (void **) malloc (m * sizeof(void *));
   for (i = 0; i < m; i++) {
      (*a)[i] = storage + i * n * size;
   }
}
```

Figure 8.22 (contd.) C program implementing dynamic programming algorithm to find an optimal binary search tree.

Use the partitioning-communication-agglomeration-mapping design methodology to implement a parallel version of this program. (Hint: You need to find an agglomeration that will allow multiple processes to be computing concurrently.)

CHAPTER

9

Document Classification

It is impossible to enjoy idling thoroughly unless one has plenty of work to do.
Jerome Klapka Jerome, *Idle Thoughts of an Idle Fellow*

9.1 INTRODUCTION

The World Wide Web contains millions of text documents. Many questions can be answered by retrieving the right documents, but automated search engines are needed to find the documents most likely to contain relevant information. To simplify the comparison of documents with queries or against each other, practitioners often use a vector to represent the contents of a document. Each dimension of the vector represents the "fit" between the document and a concept, which may take the form of a word or phrase.

In this chapter we develop an application that reads a dictionary of key words, locates a set of text documents, reads the documents, generates a vector for each document, and writes the document vectors. In contrast to most of the problems we have examined in previous chapters, this problem is amenable to a functional decomposition. We develop a manager/worker–style parallel program to solve this problem. In the course of developing the program and discussing enhancements to it, we add the following MPI functions to our repertoire:

- `MPI_Irecv`, to initiate a nonblocking receive
- `MPI_Isend`, to initiate a nonblocking send
- `MPI_Wait`, to wait for a nonblocking communication to complete
- `MPI_Probe`, to check for an incoming message
- `MPI_Get_count`, to find the length of a message
- `MPI_Testsome`, to return information on all completed nonblocking communications

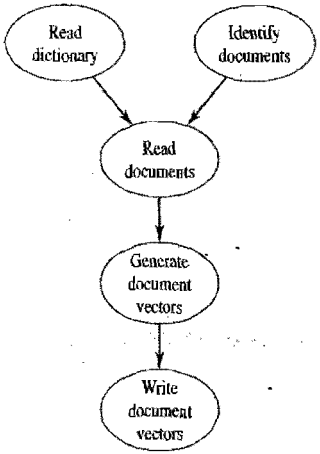


Figure 9.1 The document classification problem consists of five general tasks.

9.2 PARALLEL ALGORITHM DESIGN

Our objective is a program that reads a dictionary and searches a directory structure for plain text files (such as `.html`, `.tex`, and `.txt` files). For each of these files, the program opens the file, reads its contents, and generates a profile vector that indicates how many times the text document contains each word appearing in the dictionary. The program writes a file containing the profile vectors for each of the plain text files it has examined. A data dependence diagram for the five steps appears in Figure 9.1.

9.2.1 Partitioning and Communication

While reading the dictionary and identifying the documents may be performed concurrently, we need to break the tasks into finer pieces if we are going to exploit parallelism more fully. Let's assume that reading documents and generating the profile vectors consume the vast majority of execution time. It makes sense, then, to generate two tasks for *each document*: one to read the document file and another to generate the vector. The resulting data dependence graph appears in Figure 9.2.

This algorithm is a natural candidate for a functional decomposition. Each operation is a primitive task. A data element (i.e., a document) is associated with each task.

9.2.2 Agglomeration and Mapping

The number of tasks is not known at compile time. Tasks do not communicate with each other. The time needed to perform each task (process each document)

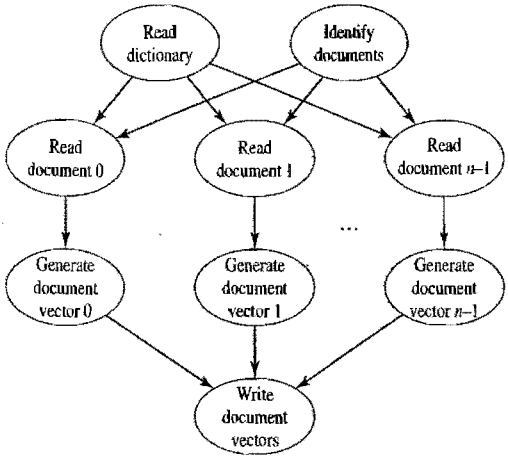


Figure 9.2 The reading and profiling of each document may occur in parallel.

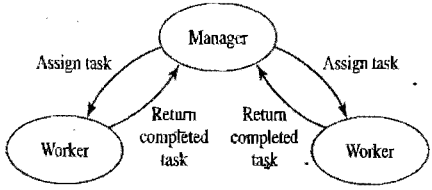


Figure 9.3 In a manager/worker–style parallel algorithm, a manager process assigns tasks to and receives results from a set of worker processes.

may vary widely, because the documents may have radically different sizes, and some documents (such as `.html` files) may be more difficult to process than others (such as `.txt` files). Given these characteristics, our mapping decision tree (Figure 3.7) suggests we should map tasks to processes at run-time.

9.2.3 Manager/Worker Paradigm

To support the run-time allocation of tasks to processes, we will construct a manager/worker–style parallel program. One process, called the **manager,** is responsible for keeping track of assigned and unassigned data. It assigns tasks to the other processes, called **workers,** and retrieves results back from them (Figure 9.3).

The advantage of allocating only a single task at a time to each worker is that it balances workloads. A worker is done when it completes a task and the

manager has no more tasks to assign. At this point, no worker has more than one task left to complete.

The disadvantage of allocating a single task at a time to each worker is that it introduces additional communication overhead into the parallel algorithm, increasing execution time and lowering speedup.

To date, all of the parallel programs we have written are in the **SPMD** (Single Program Multiple Data, sometimes pronounced "spim-dee") style. In SPMD programs every process executes the same functions (though a designated process may be responsible for file or user I/O). The manager/worker model is a clear break from the SPMD style of programming. The manager process has different responsibilities from the worker processes. In a parallel program implementing a manager/worker algorithm, there is typically a control flow split early in the program's execution that sends the manager process off executing one function and the worker processes off executing another function.

Keeping workloads balanced is essential for high efficiency, and we choose the manager/worker paradigm as the basis for our parallel algorithm design. Our first step is to decide which tasks should be done by the manager and which should be done by the workers. Identifying the documents is clearly a job for the manager, since it is the manager that will be assigning file names to the workers. Reading the dictionary should be done by the workers, since they are the processes that will be constructing the profile vectors. Given a document file name, a worker will read the file and produce the document profile vector. Finally, we'll give the manager responsibility for gathering the document vectors and writing the results file.

In the task/channel graph of Figure 9.3, you can see there is an interaction cycle between the manager and each worker. The manager provides the worker with a task. Some time later the worker returns the completed task to the manager (or simply reports that the task is done). At this point the manager may give the worker another task.

The cycle may begin with either a message from the manager to the worker or vice versa. Which should come first? In our design, we choose to have the worker initiate the dance by sending a message to the manager indicating it is ready to receive a task. We do this because we cannot be certain when the MPI processes on different processors begin execution. This way, the manager only sends tasks to workers it knows are active.

9.2.4 Manager Process

Pseudocode for the manager process appears in Figure 9.4. The manager begins by identifying the n plain text documents in the directory specified by the user. It receives from worker 0 the value of k, the number of elements in each document vector, so that it can allocate $n \times k$ matrix s for storing the vectors it receives from the workers. It initializes variables d and t showing that no documents have been assigned and no workers have been terminated, respectively.

The manager enters a loop that it repeats until it has terminated all workers. In this loop it receives a message from a worker. If the message contains a document's

```
Manager
  Local variables
    a — array showing document assigned to each process
    d — documents assigned
    j — ID of worker requesting document
    k — document vector length
    n — number of documents
    p — total number of processes (p - 1 are workers)
    s — storage array containing document vectors
    t — terminated workers
    v — individual document vector

  Identify n documents in user-specified directory
  Receive dictionary size k from worker 0
  Allocate s with dimension n × k to store document vectors
  d ← 0
  t ← 0
  repeat
    Receive message from worker j
    if message contains document vector v
      s[a[j]] ← v
    else
      {Message is first request for work—do nothing}
    endif
    if d < n then
      Send name of document d to worker j
      a[j] ← d
      d ← d + 1
    else
      Send termination message to worker j
      t ← t + 1
    endif
  until t = p - 1
  Write s to output file
```

Figure 9.4 Pseudocode for the document classification manager process.

profile vector, it stores the vector in the appropriate place in s. Otherwise, the worker is simply indicating it is ready for a document. (This only happens once per worker.) If there are any documents left, the manager sends the file name to the worker, records in array a which document it assigned, and increments d, the number of documents assigned. If there are no documents left, the manager sends a termination message to the worker and increments the termination count. It repeats the loop until it has terminated all of the workers.

Exiting the loop, the manager process writes to a file the document profile vectors stored in s.

9.2.5 Function `MPI_Abort`

Recall that after the manager identifies the n plain text documents in the directory and receives k, the document vector size, from process 0, it must allocate an $n \times k$

matrix for storing the vectors. This is an operation that only the manager process performs—the worker processes are off doing other things at this point.

If the memory allocation fails, we need a simple way to terminate the execution of the MPI program. Function `MPI_Abort` gives us this power. It has this header:

```
int MPI_Abort (MPI_Comm comm, int error_code)
```

Function `MPI_Abort` makes a "best effort" attempt to abort all processes in the communicator passed as `comm`. It returns to the calling environment the value of `error_code`.

9.2.6 Worker Process

Now let's think about the worker processes. Every worker needs a copy of the dictionary. One solution is for every worker to open the dictionary file and read its contents. Another option is for one worker to open the dictionary file, read its contents, and then broadcast the dictionary to the other workers. If the broadcast bandwidth inside the parallel computer is greater than the bandwidth between the file server and the parallel computer, the second strategy is better. It is the one we adopt.

The pseudocode for the worker process is in Figure 9.5. As soon as a worker becomes active, it notifies the manager it is ready for work. (Technically, this

```
Worker
  Local variables
    f — file name
    k — dictionary size
    v — document vector

  Send first request for work to manager
  if worker 0 then
    Read dictionary from file
  endif
  Broadcast dictionary among workers
  Build hash table from dictionary
  if worker 0 then
    Send dictionary size k to manager
  endif
  repeat
    Receive file name f from manager
    if f indicates termination
      exit loop
    else
      Read document from file f
      Generate document vector v
      Send v to manager
    endif
  forever
```

Figure 9.5 Pseudocode for the document classification worker process.

is not true, because the worker has not yet acquired the dictionary. However, making the request early allows the time spent sending the message and receiving the first document's file name from the manager to be overlapped with dictionary setup time.) Worker 0 reads the dictionary. All workers (but not the manager) participate in the collective communication operation to broadcast the dictionary. Each worker constructs a hash table from dictionary elements. This will enable constant-time access (in the average case) to dictionary entries, speeding the profiling of the documents. Worker 0 also sends the dictionary size to the manager.

The worker process enters a **repeat . . . forever** loop. It receives a message from the manager. If the message is the name of a file containing a document, the worker reads the file, generates the document vector, sends the document vector to the manager, and iterates. If the worker receives a termination message from the manager, there are no more documents to process, and the worker ceases execution.

We can create a task/channel graph for this manager/worker design. It is illustrated in Figure 9.6 for the case when there are five processes (one manager and four workers).

We need to decide which process will be the manager. It makes no particular difference, but for the sake of simplicity, let's assign management responsibilities to the process with rank 0 in `MPI_COMM_WORLD`; processes with ranks $1, 2, \ldots, p-1$ will be the workers.

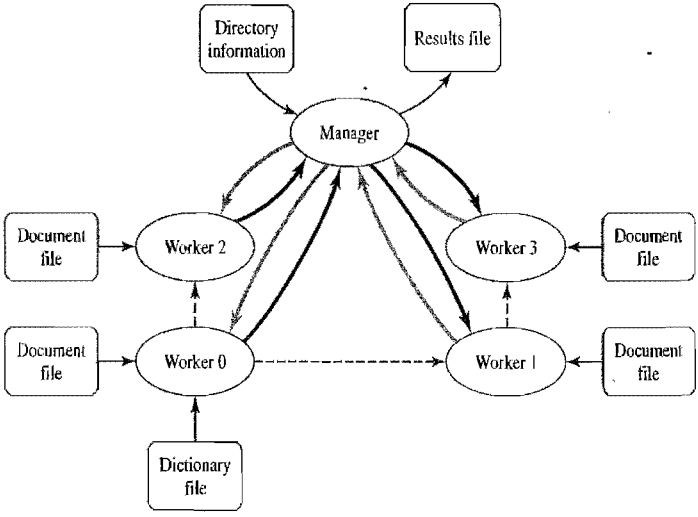


Figure 9.6 Task/channel graph for the parallel document classification algorithm. Dashed arrows represent channels used to broadcast the dictionary. Heavy gray arrows represent channels that carry document names to workers. Heavy black arrows represent channels that carry document vectors to the manager.

Note that our design assumes at least two processes will execute the program—one manager and at least one worker. Our implementation needs to check to ensure at least two MPI processes are active.

9.2.7 Creating a Workers-Only Communicator

In the parallel algorithm we have designed, the dictionary is broadcast among the workers while the manager is searching the directory structure for plain text files. Function `MPI_Bcast` is a collective communication operation, meaning it must be performed by every process in a communicator.

To support a worker-only broadcast, we must create a new communicator that includes all the workers but excludes the manager. In Chapter 8 we saw how to use `MPI_Comm_split` to split a communicator into one or more new communicators.

In this case, however, we do not want the manager process to be a member of a new communicator. We can exclude the manager process by having it pass the constant `MPI_UNDEFINED` as the value of `split_key`. The return value of `new_comm` will be `MPI_COMM_NULL`.

We can create a new, workers-only communicator with this code:

```
int id;
MPI_Comm worker_comm;
...
if (!id) /* Manager */
   MPI_Comm_split (MPI_COMM_WORLD, MPI_UNDEFINED, id,
                   &worker_comm);
else /* Worker */
   MPI_Comm_split (MPI_COMM_WORLD, 0, id, &worker_comm);
```

9.3 NONBLOCKING COMMUNICATIONS

The work of the manager process has three phases. In the first phase the manager finds the plain text files in the directory structure specified by the user, receives the dictionary size from worker 0, and allocates the two-dimensional array that is used to store the document profile vectors. In phase 2, the manager allocates documents to workers and collects profile vectors. It writes the complete set of profile vectors to a file in phase 3.

Let's focus on phase 1. The manager must search a directory structure and receive a message from worker 0. Is there a way to overlap these two activities?

To date, we have used `MPI_Send` and `MPI_Recv` for point-to-point message-passing. These are **blocking** operations. Function `MPI_Send` does not return until either the message has been copied into a system buffer or the message has been sent. In either case, you can overwrite the message buffer as soon as the function returns. Function `MPI_Recv` does not return until the message has been received into the buffer specified by the user; you may access the message values as soon as the function returns.

Blocking sends and receives may limit the performance of a parallel program. With `MPI_Send`, there may be some reason why the system does not copy the message into a system buffer. In this case the function blocks until the message has been sent, even if you have no intention of overwriting the buffer right away.

Posting a receive before a message arrives can save time, because the system can save a copy operation by transferring the contents of the incoming message directly into the destination buffer rather than a temporary system buffer. It is difficult to do this with `MPI_Recv`. If the function is called too soon, the calling process blocks until the message arrives. If the function is called too late, the incoming message has already been copied into a system buffer and must be copied again.

Fortunately, the MPI library provides **nonblocking** send and receive functions. Calls to `MPI_Isend` and `MPI_Irecv` simply post, or initiate, the appropriate communication operation. (Think of the "I" as standing for *initiate*.) The message buffer may not be accessed by the user process until it explicitly completes the communication with a call to `MPI_Wait`.

Posting a message, performing other computations or I/O operations, and then completing the message, may save time in two different ways. First, it may allow the system to eliminate message-copying by the sending and/or the receiving processes. Second, it allows dedicated communication coprocessors, if they exist on the parallel computer, to perform communication-related activities while the CPU assigned to the computation manipulates local data.

9.3.1 Manager's Communication

Getting back to our manager process, it knows at the beginning of its execution that it needs to receive the dictionary size from a worker, even though it does not actually use this value until after it has identified the document files to be processed. (Of course, the value of a nonblocking read is higher when the length of the message to be received is greater.) Let's look at the two MPI functions needed to perform a nonblocking receive.

9.3.2 Function `MPI_Irecv`

Function `MPI_Irecv` has this header:

```
int MPI_Irecv (void *buffer, int cnt, MPI_Datatype dtype,
        int src, int tag, MPI_Comm comm, MPI_Request *handle)
```

The first six parameters are identical to those of `MPI_Recv`. However, since `MPI_Irecv` only initiates the receive, you cannot access `buffer` until a matching call to `MPI_Wait` has returned. The function returns, through the last parameter, a **handle** (pointer) to an `MPI_Request` object that identifies the communication operation that has been initiated.

Note that the function does not return a pointer to an `MPI_Status` object, since the receive has not yet been completed.

9.3.3 Function `MPI_Wait`

Here is the header for function `MPI_Wait`:

```
int MPI_Wait (MPI_Request *handle, MPI_Status *status)
```

Function `MPI_Wait` blocks until the operation associated with pointer `handle` completes. In the case of a send operation, the buffer may then be assigned new values. In the case of a receive operation, the buffer may be referenced, and `status` points to the `MPI_Status` object containing information about the received message.

9.3.4 Workers' Communications

Now let's examine the needs of the worker processes for new MPI functionality. Before being assigned its first document, each worker must notify the manager process that it is active. It can initiate this send to the manager, then proceed immediately to the broadcasting of the dictionary and the construction of the hash table.

The worker also must receive file names (actually, complete path names) from the manager. There is no way of knowing in advance how long these names may be, since directory structures may be deeply nested. For this reason it would be convenient if the worker could check on an incoming message and determine its length before actually reading it.

Here are the three MPI functions that meet these needs of the workers.

9.3.5 Function `MPI_Isend`

```
int MPI_Isend (void *buffer, int cnt, MPI_Datatype dtype,
      int dest, int tag, MPI_Comm comm, MPI_Request *handle)
```

Function `MPI_Isend` posts a nonblocking send operation. The first six parameters have the same meaning as in `MPI_Send`. The last parameter is an output parameter—a handle to an opaque `MPI_Request` object created by the run-time system. It identifies this communication request. The message buffer may not be reused until the matching call to `MPI_Wait` has returned.

9.3.6 Function `MPI_Probe`

```
int MPI_Probe (int src, int tag, MPI_Comm comm,
               MPI_Status *status)
```

Passed `src`, the rank of the message source; `tag`, the incoming message's tag; `comm`, the communicator; and `status`, a pointer to an `MPI_Status` object, function `MPI_Probe` blocks until a message matching the source and tag specifications is available to be received. It returns through the `status` pointer information about the source, tag, and length of the message, but it does not actually receive the message.

By passing MPI_ANY_SOURCE as the src argument, you can probe for a message from any other process. Passing MPI_ANY_TAG as the tag argument allows you to probe for any message from the process you specified in src. Using both MPI_ANY_SOURCE and MPI_ANY_TAG will allow the probe to match any sent message.

In general, it is best to keep the source and tag specifications as narrow as possible, to minimize mismatch bugs that occur when messages arrive in an unexpected order. In this case, the worker knows both the source and the tag of the message it is expecting from the manager, and there is no need for it to use these constants.

9.3.7 Function MPI_Get_count

```
int MPI_Get_count (MPI_Status *status, MPI_Datatype dtype,
                   int *cnt)
```

Passed status, a pointer to an MPI_Status object; dtype, the data type of the message elements; and cnt, a pointer to an integer, function MPI_Get_count returns through cnt the number of elements in the message.

9.4 DOCUMENTING THE PARALLEL PROGRAM

With these new MPI functions in hand, we may now construct a parallel program to perform the document classification task. We do not include the entire program in this section; we omit the directory-searching and profile-writing functions called by the manager process and the hash-table-building and profile-generating functions called by the worker processes. Our focus is on the general structure of a manager/worker MPI program and how the new MPI functions introduced in this chapter fit into the overall design.

The program appears in Figure 9.7.

We define four constants to be used as tags for the four types of messages the processes are sending and receiving.

Using message tags helps document the code. It also allows a process to receive messages from another process in a different order than they were sent.

For example, worker 0—like all workers—sends an initial request for work to the manager. After it has read, broadcast, and processed the dictionary, worker 0 sends the dictionary size to the manager. The manager, on the other hand, needs to construct the document vector profile storage area before it begins handling requests for work from processes. For this reason it wants to receive the dictionary size message from worker 0 before it receives worker 0's initial request for work. We give these two messages different tags, enabling their out-of-order reception. We use DICT_SIZE_MSG as the tag for the message from worker 0 to the manager that contains the number of words in the dictionary file. It and all other workers inform the manager that they are active by sending the manager an empty message with the tag EMPTY_MSG.

```
/*
 *   Document Classification Program
 */

#include <mpi.h>
#include <stdio.h>
#include <string.h>
#include <sys/stat.h>
#include <ftw.h>

#define DICT_SIZE_MSG   0  /* Msg has dictionary size */
#define FILE_NAME_MSG   1  /* Msg is file name */
#define VECTOR_MSG      2  /* Msg is profile */
#define EMPTY_MSG       3  /* Msg is empty */

#define DIR_ARG         1  /* Directory argument */
#define DICT_ARG        2  /* Dictionary argument */
#define RES_ARG         3  /* Results argument */

typedef unsigned char uchar;

int main (int argc, char *argv[]) {

   int         id;           /* Process rank */
   int         p;            /* Number of processes */
   MPI_Comm    worker_comm;  /* Workers-only communicator */

   void manager (int, char **, int);
   void worker (int, char **, MPI_Comm);

   MPI_Init (&argc, &argv);
   MPI_Comm_rank (MPI_COMM_WORLD, &id);
   MPI_Comm_size (MPI_COMM_WORLD, &p);

   if (argc != 4) {
      if (!id) {
         printf ("Program needs three arguments:\n");
         printf ("%s <dir> <dict> <results>\n", argv[0]);
      }
   } else if (p < 2) {
      printf ("Program needs at least two processes\n");
   } else {
      if (!id) {
         MPI_Comm_split (MPI_COMM_WORLD, MPI_UNDEFINED,
            id, &worker_comm);
         manager (argc, argv, p);
      } else {
         MPI_Comm_split (MPI_COMM_WORLD, 0, id, &worker_comm);
         worker (argc, argv, worker_comm);
      }
   }
   MPI_Finalize();
   return 0;
}
```

Figure 9.7 Document classification program.

```
void manager (int argc, char *argv[], int p) {

   int           assign_cnt; /* Docs assigned so far */
   int          *assigned;   /* Document assignments */
   uchar        *buffer;     /* Store profile vectors here */
   int           dict_size;  /* Dictionary entries */
   int           file_cnt;   /* Plain text files found */
   char        **file_name;  /* Stores file (path) names */
   int           i;
   MPI_Request   pending;    /* Handle for recv request */
   int           src;        /* Message source process */
   MPI_Status    status;     /* Message status */
   int           tag;        /* Message tag */
   int           terminated; /* Count of terminated procs */
   uchar       **vector;     /* Profile vector repository */

   void build_2d_array (int, int, uchar ***);
   void get_names (char *, char ***, int *);
   void write_profiles (char *, int, int, char **, uchar **);

   /* Put in request to receive dictionary size */
   MPI_Irecv (&dict_size, 1, MPI_INT, MPI_ANY_SOURCE,
      DICT_SIZE_MSG, MPI_COMM_WORLD, &pending);

   /* Collect the names of the documents to be profiled */
   get_names (argv[DIR_ARG], &file_name, &file_cnt);

   /* Wait for dictionary size to be received */
   MPI_Wait (&pending, &status);

   /* Set aside buffer to catch profiles from workers */
   buffer = (uchar *)
      malloc (dict_size * sizeof(MPI_UNSIGNED_CHAR));

   /* Set aside 2-D array to hold all profiles.
      Call MPI_Abort if the allocation fails. */
   build_2d_array (file_cnt, dict_size, &vector);

   /* Respond to requests by workers. */
   terminated = 0;
   assign_cnt = 0;
   assigned = (int *) malloc (p * sizeof(int));

   do {
      /* Get profile from worker */
      MPI_Recv (buffer, dict_size, MPI_UNSIGNED_CHAR,
         MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD,
         &status);
      src = status.MPI_SOURCE;
      tag = status.MPI_TAG;
      if (tag == VECTOR_MSG) {
         for (i = 0; i < dict_size; i++)
            vector[assigned[src]][i] = buffer[i];
      }
```

Figure 9.7 (contd.) Document classification program.

```
      /* Assign more work or tell worker to stop. */
      if (assign_cnt < file_cnt) {
         MPI_Send (file_name[assign_cnt],
            strlen(file_name[assign_cnt])+1,
            MPI_CHAR, src, FILE_NAME_MSG, MPI_COMM_WORLD);
         assigned[src] = assign_cnt;
         assign_cnt++;
      } else {
         MPI_Send (NULL, 0, MPI_CHAR, src, FILE_NAME_MSG,
            MPI_COMM_WORLD);
         terminated++;
      }
   } while (terminated < (p-1));

   write_profiles (argv[RES_ARG], file_cnt, dict_size,
      file_name, vector);
}

void worker (int argc, char *argv[], MPI_Comm worker_comm)
{
   char          *buffer;     /* Words in dictionary */
   hash_el      **dict;       /* Hash table of words */
   int            dict_size;  /* Profile vector size */
   long           file_len;   /* Chars in dictionary */
   int            i;
   char          *name;       /* Name of plain text file */
   int            name_len;   /* Chars in file name */
   MPI_Request    pending;    /* Handle for MPI_Isend */
   uchar         *profile;    /* Document profile vector */
   MPI_Status     status;     /* Info about message */
   int            worker_id;  /* Rank in worker_comm */

   void build_hash_table (char *, int, hash_el ***, int *);
   void make_profile (char *, hash_el **, int, uchar *);
   void read_dictionary (char *, char **, long *);

   /* Worker gets its worker ID number */

   MPI_Comm_rank (worker_comm, &worker_id);

   /* Worker makes initial request for work */

   MPI_Isend (NULL, 0, MPI_UNSIGNED_CHAR, 0, EMPTY_MSG,
              MPI_COMM_WORLD, &pending);

   /* Read and broadcast dictionary file */

   if (!worker_id)
      read_dictionary (argv[DICT_ARG], &buffer, &file_len);
   MPI_Bcast (&file_len, 1, MPI_LONG, 0, worker_comm);
   if (worker_id) buffer = (char *) malloc (file_len);
   MPI_Bcast (buffer, file_len, MPI_CHAR, 0, worker_comm);
```

Figure 9.7 (contd.) Document classification program.

```
        /* Build hash table */

        build_hash_table (buffer, file_len, &dict, &dict_size);

        profile = (uchar *) malloc (dict_size * sizeof(uchar));

        /* Worker 0 sends msg to manager re: size of dictionary */

        if (!worker_id) MPI_Send (&dict_size, 1, MPI_INT, 0,
                          DICT_SIZE_MSG, MPI_COMM_WORLD);

        for (;;) {

           /* Find out length of file name */

           MPI_Probe (0, FILE_NAME_MSG, MPI_COMM_WORLD, &status);
           MPI_Get_count (&status, MPI_CHAR, &name_len);

           /* Drop out if no more work */

           if (!name_len) break;

           name = (char *) malloc (name_len);
           MPI_Recv (name, name_len, MPI_CHAR, 0, FILE_NAME_MSG,
                     MPI_COMM_WORLD, &status);

           make_profile (name, dict, dict_size, profile);
           free (name);

           MPI_Send (profile, dict_size, MPI_UNSIGNED_CHAR, 0,
                     VECTOR_MSG, MPI_COMM_WORLD);
        }
     }
```

Figure 9.7 (contd.) Document classification program.

When the manager assigns a document to a worker, it uses message tag `FILE_NAME_MSG`. When a worker responds with the profile vector for that document, it uses message tag `VECTOR_MSG`.

We also define constants to refer to the three command-line arguments. The first argument, indexed by `DIR_ARG`, is the name of the directory that serves as the root of the directory structure to be searched for plain text files. The second, `DICT_ARG`, is the name of the file containing the dictionary. The third argument, indexed by `RES_ARG`, is the name of the output file that contains the set of document profile vectors upon successful completion of the program.

Function `main` contains code that all processes execute before the manager goes one way and the workers go another. Execution begins with the traditional calls to initialize MPI and retrieve the process rank and the process group size.

The function checks to ensure that the user supplied the correct number of arguments on the command line. If not, execution will not continue. The function also checks to ensure that there are at least two processes. Without at least one worker, no documents will be processed. If these conditions are satisfied, all

processes cooperate to create a new, workers-only communicator. After this has been done, the roles of the manager and the workers diverge. Process 0 (the manager) calls `manager` and the other processes call `worker`.

Only a single process executes function `manager`. Refer back to the pseudocode of Figure 9.4 for a refresher on how it is structured. The manager begins by posting a receive for the message containing the size of the dictionary. It then calls `get_names` to construct `file_name`, an array of strings. These strings are the names of the plain text files in the directory tree specified by the user on the command line. After the function returns, `file_cnt` is the number of files that need to be processed.

At this point the manager needs to allocate the memory that will be used to hold the document profile vectors. The number of vectors is equal to the number of documents (`file_cnt`). The length of the vectors depends on the number of dictionary entries. So the manager must wait until the receive it posted has been completed. After the message containing the dictionary size has arrived, the manager constructs the two-dimensional array holding the vectors.

Before the principal loop of the function, the manager initializes the number of terminated processes and the number of assigned documents to 0. It also allocates the array that will be used to keep track of the document currently assigned to each process.

Inside the loop, the manager receives the next message from a worker. If the message tag indicates the message contains a document profile vector, the manager stores it. If unassigned documents remain, the manager sends the name of the next unassigned document to the worker and increments the number of assigned documents. Otherwise, it sends an empty file name to the worker, indicating to the worker that it should cease, and increments the number of terminated workers. The loop continues until all of the workers have been terminated.

The manager exits the loop only after it has received all of the document profile vectors from the workers. It writes the vectors to the file the user specified on the command line.

Now let's look at function `worker`. If you need to refresh your memory of what the worker does, refer to the pseudocode in Figure 9.5. Each worker begins by finding its rank in the worker-only communicator. If the workers did not interact, this would not be necessary, but in this algorithm, worker 0 is responsible for reading the dictionary file and broadcasting it to the other workers. Hence the workers need to know their ranks.

After calling `MPI_Comm_rank`, each worker makes its initial request for work. The message tag `EMPTY_MSG` indicates to the manager that this is the worker's initial request for work, not a message containing a document profile vector.

Next, worker 0 reads the dictionary and broadcasts it to the other workers. Note that before broadcasting the dictionary worker 0 broadcasts an integer containing the size of the dictionary. That way, the other workers can allocate enough space to hold the dictionary's contents. The workers extract the words from the dictionary and put them in a hash table. This will speed the document classification

task by enabling the process to determine in constant time (on average) if a word in the document appears in the dictionary.

After extracting words from the dictionary, the workers know how large the document profile vectors will be. In this implementation the document profile vector contains an unsigned character for each dictionary entry, enabling the correlation between the document and that entry to be expressed as an integer between 0 and 255. Each worker allocates room for a profile vector. Worker 0 sends the dictionary-size message to the manager.

Now the worker enters its principal loop. It probes for a message from the manager containing the name of a plain text document file. It allocates enough room to receive the file name, then calls `MPI_Recv` to actually get the name. Given the file name and the hash table, function `make_profile` builds the document profile vector. The worker sends this vector back to the manager. When the worker receives a zero-length file name from the manager, that means there are no more documents to process, and the worker returns from the function.

9.5 ENHANCEMENTS

In this section we consider ways to improve the execution time of our parallel document classification program.

9.5.1 Assigning Groups of Documents

In some applications a preallocation of data to processes can result in an imbalanced workload. Imbalanced workloads lead to idle processors, which lowers speedup. Allocating data to processes at run-time balances the workloads. On the other hand, it introduces additional interprocessor communication overhead which lowers speedup. Sometimes, the best design chooses a middle point between the two extremes. For example, we might construct a manager/worker algorithm in which the manager assigns k tasks at a time to workers.

9.5.2 Pipelining

Let's reconsider the task graph of Figure 9.2. If one process can retrieve the dictionary file from the file server as quickly as k tasks, then there is not much we can do with that task. We're already allowing processes to build their hash tables concurrently. We will consider this task no further.

On the other hand, we can make improvements to the document identification and results writing tasks. When we began our design, we assumed that nearly all of the time would be spent reading document files and generating the associated profile vectors. We left the tasks of identifying the plain text files and writing the results file as sequential tasks. No document files are processed until the manager has identified all of them. If the time needed to perform these tasks is not negligible, then our design will not scale well to larger numbers of processes (Amdahl's Law).

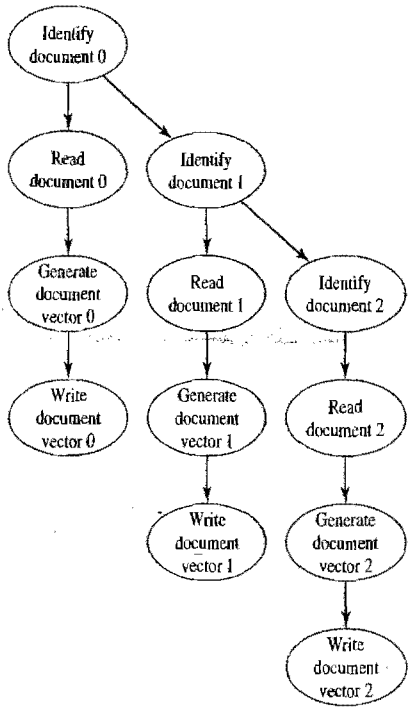


Figure 9.8 By dividing the task of identifying documents into its elemental tasks, we can expose the opportunity for pipelining the processing of the documents. We can be constructing the profile vector for document 0 while we are reading document 1 while we are identifying document 2.

Let's reconsider the task graph of Figure 9.2, ignoring the dictionary construction task. What happens when we divide the document identification task into smaller units? We end up with the new task graph shown in Figure 9.8. If we "dangle" the graph by the "Identify Document 0" node, we can see that while we are identifying document 1, we could begin reading document 0. While we are in one phase of processing document i, we can be in later phases of processing documents $i-1$, $i-2$, etc. This is an example of pipelining.

Pipelining can dramatically reduce the execution time of a parallel algorithm exhibiting functional parallelism. For example, Figure 9.9 illustrates how a pipelined program with one read process, two worker processes, and one write process could outperform a nonpipelined program with one manager and three workers.

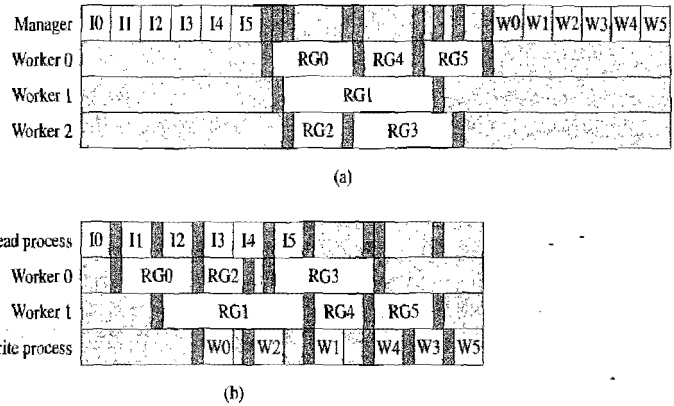


Figure 9.9 Pipelining tasks can reduce the execution time of programs with functional parallelism. In this figure I*i* refers to the task of identifying text file *i*, RG*i* refers to the task of reading file *i* and generating document profile vector *i*, and W*i* refers to the task of writing document profile vector *i* to a file. The dark gray bars represent time spent communicating. (a) A manager process identifies all text files before assigning them to workers. It collects all document vectors before writing them. This is the approach taken for the program developed in this chapter. (b) A pipelined solution. One process identifies text files, two more processes read text files and generate profile vectors, and a fourth task writes the profile vectors to a file.

The downside of implementing a manager/worker program incorporating pipelining is that it can be much more complicated. In the previous implementation the manager identified all the document file names before responding to any worker requests. Suppose we want to implement a manager that gets workers busy as soon as possible. In other words, as soon as the manager has identified at least one document and has received at least one request for work from a process, it starts sending document names to processes. That means the manager must multiplex its time between identifying documents and responding to the requests of the workers.

Here is one way we might implement the document identification/task assignment logic. Let j be the number of unassigned tasks and w be the number of workers waiting for something to do. If $j > 0$ and $w > 0$, then the manager can assign $\min(j, w)$ tasks to workers. If $j > 0$, the manager should check to see if any messages from workers have arrived. If so, the manager can receive these messages. Then we're back in the situation where $j > 0$ and $w > 0$. Otherwise, the manager should find more tasks.

9.5.3 Function `MPI_Testsome`

To implement this functionality, we need a way to check, without blocking, whether one or more expected messages have arrived. The MPI library provides four functions to do this: `MPI_Test`, `MPI_Testall`, `MPI_Testany`, and

MPI_Testsome. All of these functions require that you pass them handles to MPI_Request objects that result from calls to nonblocking receive functions. We'll describe how to use MPI_Testsome, which is the best function to use for the purpose we have described.

The manager posts a nonblocking receive to each of the worker processes. It builds an array of handles to the MPI_Request objects returned from these function calls. In order to determine if messages have arrived from any or all of the workers, the manager calls MPI_Testsome, which returns information about how many of the messages have arrived.

Function MPI_Testsome has this header:

```
int MPI_Testsome (int in_cnt, MPI_Request *handlearray,
                  int *out_cnt, int *index_array,
                  MPI_Status *status_array)
```

Passed in_cnt, the number of nonblocking receives to check, and handlearray, an array containing MPI_Request handles, function MPI_Testsome returns out_cnt, the number of completed communications. The first out_cnt entries of index_array contain the indices in handlearray of the completed communications. The first out_cnt entries of status_array contain the status records for the completed communications.

9.6 SUMMARY

The manager/worker paradigm is an effective way to think of parallel computations where it is difficult to preallocate work to processes and guarantee balanced workloads. In this chapter we considered the problem of classifying a set of plain text documents according to a user-supplied dictionary. Since document sizes can vary widely, and since some documents may be easier to process than others, the manager/worker design is appropriate.

In the process of developing this application, we introduced some additional MPI capabilities. We used function MPI_Comm_split to create a new, workers-only communicator that facilitated the broadcast of the dictionary among the workers. We discovered several places where communications could be overlapped with either computations or other I/O operations. We introduced the nonblocking communications functions MPI_Isend and MPI_Irecv and their companion completion function, MPI_Wait. We also saw how it could be beneficial for the worker processes to check the length of the path names sent by the manager before actually reading them. Functions MPI_Probe and MPI_Get_count allow this to be done.

We examined two ways to enhance the performance of the parallel program. The first enhancement is to consider a "middle ground" between preallocating all documents to tasks (which can lead to an imbalanced workload) and allocating documents one at a time to tasks (which can lead to excessive interprocessor communication). In some applications the best performance may be obtained by allocating small groups of tasks to workers.

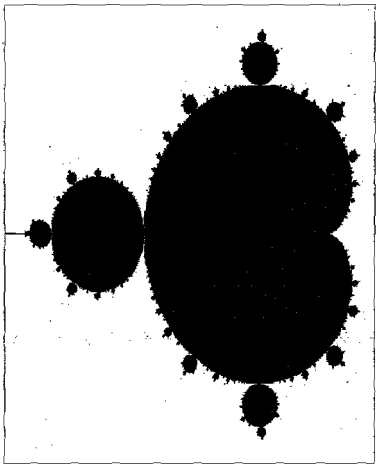


Figure 9.10 The Mandelbrot set is an example of a fractal. In this figure the lower left corner of the box represents the complex number $-1.5 - i$. The upper right corner of the box represents the complex number $0.5 + i$. Black points are in the set.

The magnitude of z is its distance from the origin; i.e., the length of the vector formed by its real and imaginary parts. If $z = a + bi$, the magnitude of z is $\sqrt{a^2 + b^2}$. If the magnitude of z ever becomes greater than or equal to 2, its subsequent values will grow without bound, and we know that c is not a point in the Mandelbrot set. If we iterate n times and find that the magnitude of z_n is still less than 2, we can conclude c is in the Mandelbrot set.

Your program should compute a Mandelbrot set for 600×600 evenly spaced points in a square region of the complex plane bounded by $-1.5 - i$ and $1 + i$. Let $n = 1{,}000$. If $z_{1000} < 2$, you should display point c as a member of the Mandelbrot set.

9.10 A **perfect number** is a positive integer whose value is equal to the sum of all its positive factors, excluding itself. The first two perfect numbers are 6 and 28:

$$6 = 1 + 2 + 3$$

$$28 = 1 + 2 + 4 + 7 + 14$$

The Greek mathematician Euclid (c. 300 BCE) showed that if $2^n - 1$ is prime, then $(2^n - 1)2^{n-1}$ is a perfect number. For example, $2^2 - 1 = 3$ is prime, so $(2^2 - 1)2^1 = 6$ is a perfect number. Write a parallel program to find the first eight perfect numbers.

CHAPTER

10

Monte Carlo Methods

O! many a shaft at random sent
Finds mark the archer little meant!
And many a word, at random spoken,
May soothe or wound a heart that's broken!
Sir Walter Scott, *The Lord of the Isles*

10.1 INTRODUCTION

A **Monte Carlo method** is an algorithm that solves a problem through the use of statistical sampling. The name is derived from the resort city in Monaco, famous for its games of chance. While early work in this field began in the nineteenth century, the first important use of Monte Carlo methods was for the development of the atomic bomb during World War II.

The Monte Carlo method is the only practical way to evaluate integrals of arbitrary functions in six or more dimensions. It has many other uses, including predicting the future level of the Dow Jones Industrial Average, solving partial differential equations, sharpening satellite images, modeling cell populations, and finding approximate solutions to NP-hard problems in polynomial time.

To illustrate the Monte Carlo method, let's begin with a physical analogy. Suppose we want to compute the value of π. We know that the area of a circle with diameter D is $\pi D^2/4$. We also know that the area of a square having sides of length D is D^2. Imagine slipping a round cake pan with diameter D inside a $D \times D$ cake pan and putting the pans out in the rain. After a few hours, we retrieve the pans and measure the amount of water in each. The ratio of the amount of water collected in the round pan to the total amount of water collected in both pans should be about $\pi/4$:

$$\frac{\pi D^2/4}{D^2} = \frac{\pi}{4}$$

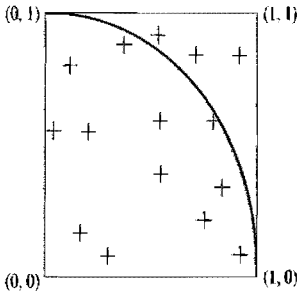


Figure 10.1 Using the Monte Carlo method to estimate the value of π. The area inside the quarter circle is $pi/4$. In this illustration, 12 of 15 points randomly chosen from the unit square are inside the circle, resulting in an estimate of 0.8 for $\pi/4$ or 3.2 for π.

We can use random numbers to perform a similar estimation. (This example shows the methodology, but keep in mind numerical integration is a better strategy when the number of dimensions is small.) Figure 10.1 illustrates a quarter circle with radius 1 embedded in a unit square. A complete circle with radius 1 has area π; hence the area of the quarter circle is $\pi/4$. We will generate a series of pairs (x, y), where both x and y are taken from a uniform random distribution between 0 and 1. Each pair represents a point inside the unit square. We keep track of the fraction f of points falling inside the quarter circle; that is, the points for which $x^2 + y^2 \leq 1$. Since $f \approx \pi/4$, we know $4f \approx \pi$.

We have implemented a C program to compute π using this methodology (Figure 10.2). Table 10.1 shows how the absolute error between the computed value of π and the actual value slowly decreases as the sample size n increases. (Given estimated value e and correct value a, the absolute error is $|e - a|/a$.) The function $1/(2\sqrt{n})$ closely approximates the absolute error of this Monte Carlo method.

10.1.1 Why Monte Carlo Works

The mean value theorem states that

$$I = \int_a^b f(x)\,dx = (b-a)\bar{f}$$

where $\bar{f}$ represents the mean (average) value of $f(x)$ in the interval $[a, b]$. (See Figure 10.3.)

The Monte Carlo method estimates the value of I by evaluating $f(x_i)$ at n points selected from a uniform random distribution over $[a, b]$. The expected

Table 10.1 As the sample size increases, so does the accuracy of the estimated solution.

Sample size n	Estimate of π	Error	$1/(2\sqrt{n})$
10	2.40000	0.23606	0.15811
100	3.36000	0.06952	0.05000
1,000	3.14400	0.00077	0.01581
10,000	3.13920	0.00076	0.00500
100,000	3.14132	0.00009	0.00158
1,000,000	3.14006	0.00049	0.00050
10,000,000	3.14136	0.00007	0.00016
100,000,000	3.14154	0.00002	0.00005
1,000,000,000	3.14155	0.00001	0.00002

```
/*
 *   This C program uses the Monte Carlo method to
 *   compute the value of pi.
 */

#include <stdio.h>
#include <stdlib.h>

int main (int argc, char *argv[])
{
   int    count;          /* Points inside circle */
   int    i;
   int    n;              /* Number of samples */
   double pi;             /* Estimate of pi */
   unsigned short xi[3]; /* Random number seed */
   double x, y;           /* Point's coordinates */

   if (argc != 5) {
      printf ("Correct command line: ");
      printf ("%s <# samples> <seed0> <seed1> <seed2>\n",
         argv[0]);
      return -1;
   }
   n = atoi(argv[1]);
   for (i = 0; i < 3; i++)
      xi[i] = atoi(argv[i+2]);

   count = 0;
   for (i = 0; i < n; i++) {
      x = erand48(xi);
      y = erand48(xi);
      if (x*x+y*y <= 1.0) count++;
   }
   pi = 4.0 * (double) count / (double) n;
   printf ("Samples: %d   Estimate of pi: %7.5f\n", n, pi);
}
```

Figure 10.2 A C program computing π using the Monte Carlo method. The precision of the answer is related to the number of samples and the quality of the pseudo-random number generator.

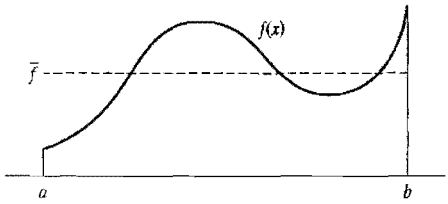


Figure 10.3 By the mean value theorem we know that the area under curve $f(x)$ is identical to the area under $\bar{f}$, where $\bar{f}$ is the mean value of $f(x)$ in the interval $[a, b]$.

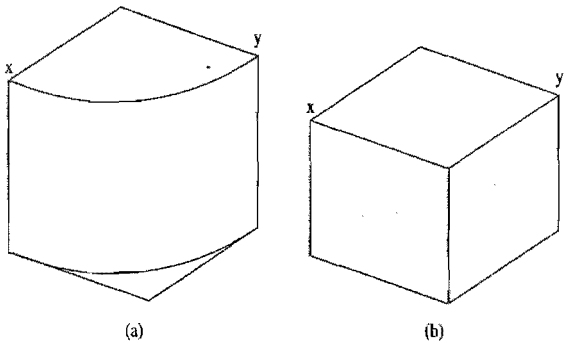


Figure 10.4 By the mean value theorem we know the volumes beneath both these surfaces are identical. (a) Within the square bounded by $0 \leq x, y \leq 1$, the height of the surface is 1 where $x^2 + y^2 \leq 1$ and 0 otherwise. (b) Within the square bounded by $0 \leq x, y \leq 1$, the height of the surface is $\pi/4$.

value of $\frac{1}{n}\sum_{i=0}^{n-1} f(x_i)$ is $\bar{f}$. Hence

$$I = \int_a^b f(x)\,dx = (b-a)\bar{f} \approx (b-a)\frac{1}{n}\sum_{i=0}^{n-1} f(x_i)$$

Let's see how this applies to the π estimation algorithm we've already described. We know that the ratio in area between a quarter circle of radius 1 and a square having sides of length 1 is $\pi/4$. Consider the surface illustrated in Figure 10.4a. This surface has height 1 if $x^2 + y^2 \leq 1$ and 0 otherwise. If we set an accumulator to 0, randomly generate pairs of points x, y from the unit square, add 1 to the accumulator if $x^2 + y^2 \leq 1$ and add nothing to the accumulator if $x^2 + y^2 > 1$, we are sampling from this surface.

If, after n samples, we divide the contents of the accumulator by n, we produce a mean. The expected value of the calculated mean is $\pi/4$, as illustrated in Figure 10.4b. Hence multiplying the calculated mean by 4 yields a Monte Carlo estimate of π.

Importantly, the error in the Monte Carlo estimate of I decreases by a factor of $1/\sqrt{n}$. *The rate of convergence is independent of the dimension of the integrand.* This is in sharp contrast to deterministic numerical integration methods, such as Simpson's rule, which have a rate of convergence that decreases as the dimension increases. It explains why Monte Carlo techniques are superior to deterministic numerical integration methods when the integrand has more than about six dimensions.

10.1.2 Monte Carlo and Parallel Computing

Monte Carlo algorithms often migrate easily onto parallel systems. Many parallel Monte Carlo programs have a negligible amount of interprocessor communications. When this is the case, p processors can be used either to find an estimate about p times faster or to reduce the error of the estimate by a factor of $\sqrt{p}$. Another way of expressing the second point is to say that p processes can reduce the variance of the answer by a factor of p.

Of course, these levels of improvement are based on the assumption that the random numbers are statistically independent. A principal challenge in the development of parallel Monte Carlo methods has been the development of good parallel random number generators. It is widely claimed that half of all supercomputer cycles are dedicated to Monte Carlo calculations. For that reason it's important to understand what makes a good parallel random number generator. We will start with a quick overview of sequential random number generators.

10.2 SEQUENTIAL RANDOM NUMBER GENERATORS

Technically, the random number generators you'll find on today's computers are **pseudo-random number generators,** because their operation is deterministic, and hence the sequences they produce are predictable. In the best case these sequences are a reasonable approximation of a truly random sequence. However, since "pseudo-random number generator" is a mouthful, we'll stick with the simpler phrase. In the remainder of this chapter, when you see the phrase "random number generator," understand we're talking about a pseudo-random number generator.

Coddington [17] has identified ten properties of the sequence of numbers produced by an ideal random number generator:

- It is **uniformly distributed,** meaning each possible number is equally probable.
- The numbers are uncorrelated.

- It never cycles; that is, the numbers do not repeat themselves.
- It satisfies any statistical test for randomness.
- It is reproducible.
- It is machine-independent; that is, the generator produces the same sequence of numbers on any computer.
- It can be changed by modifying an initial "seed" value.
- It is easily split into many independent subsequences.
- It can be generated rapidly.
- It requires limited computer memory for the generator.

There are no random number generators that meet all of these requirements. For example, computers rely upon finite precision arithmetic. Hence the random number generator may take on only a finite number of states. Eventually it must enter a state it has been in previously, at which point it has completed a cycle and the numbers it produces will begin to repeat themselves. The **period** of a random number generator is the length of this cycle.

Similarly, since we demand that the sequence of numbers be reproducible, the numbers cannot be completely uncorrelated. The best we can hope for is that the correlations be so small that they have no appreciable impact on the results of the computation.

There is often a trade-off between the speed of a random number generator and the quality of the numbers it produces. Since the time needed to generate a random number is typically a small part of the overall computation time of a program, speed is much less important than quality.

In the following sections we consider two important classes of random number generators: linear congruential and lagged Fibonacci.

10.2.1 Linear Congruential

The linear congruential method is more than 50 years old, and it is still the most popular. **Linear congruential generators** produce a sequence X_i of random integers using this formula:

$$X_i = (a \times X_{i-1} + c) \bmod M$$

where a is called the multiplier, c is called the additive constant, and M is called the modulus. In some implementations $c = 0$. When $c = 0$, it is called **a multiplicative congruential generator.** All three values must be carefully chosen in order to ensure that the sequence has a long period and good randomness properties. The maximum period is M. For 32-bit integers the maximum period is 2^{32}, or about 4 billion. This is too small a period for modern computers that execute billions of instructions per second. A quality generator has 48 bits of precision or more.

The particular sequence of integer values produced by the generator depends on the initial value X_0, which is called the **seed.** Typically the user provides the seed value.

Linear congruential methods may also be used to generate floating-point numbers. Since the generator produces integers between 0 and $M-1$, dividing X_i by M produces a floating-point number x_i in the interval $[0, 1)$.

The defects of linear congruential generators are well documented. The least significant bits of the numbers produced are correlated. (This is particularly true when the modulus M is a power of 2.) If you produce a scatter plot of ordered tuples $(x_i, x_{i+1}, \ldots, x_{i+k-1})$ in a k-dimensional unit hypercube, you'll see a lattice structure [79]. Since this problem becomes more pronounced as the number of dimensions increases, it can affect the quality of high-dimensional simulations relying on a linear congruential random number generator.

Despite these flaws, linear congruential generators with 48 or more bits of precision and carefully chosen parameters "work very well for all known applications, at least on sequential computers" [17].

10.2.2 Lagged Fibonacci

The popularity of lagged Fibonacci generators is rising, because they are capable of producing random number sequences with astonishingly long periods, while also being fast. The method produces a sequence of X_is. Each element is defined as follows:

$$X_i = X_{i-p} \star X_{i-q}$$

where p and q are the lags, $p > q$, and $\star$ is any binary arithmetic operation. Examples of suitable $\star$ operations are addition modulo M, subtraction modulo M, multiplication modulo M, and bitwise exclusive or. In the case of addition and subtraction, the X_is may be either integers or floating-point numbers. If the sequence contains floating-point numbers, $M = 1$. If $\star$ is multiplication, the sequence must consist solely of odd integers.

Note that unlike linear congruential generators, which require only a single seed value, lagged Fibonacci generators require p seed values $X_0, X_1, \ldots, X_{p-1}$. Careful selection of p, q, and M, as well as $X_0, \ldots, X_{p-1}$ results in sequences with very long periods and good randomness. If the X_is have b bits, the maximum periods attainable are $2^p - 1$ for exclusive or, $(2^p - 1)2^{b-1}$ for addition and subtraction, and $(2^p - 1)2^{b-3}$ for multiplication. Notice that increasing the maximum lag p increases the storage requirements but also increases the maximum period.

Function random, callable from C, is an additive lagged Fibonacci generator with a default lag of 31. Coddington reports this lag is much too small. He recommends setting (p, q) to at least (1279, 1063).

10.3 PARALLEL RANDOM NUMBER GENERATORS

Parallel Monte Carlo methods depend upon our ability to generate a large number of high-quality random number sequences. In addition to the properties mentioned in the previous section for sequential random number generators, an ideal parallel

random number generator would have these properties:

- No correlations among the numbers in different sequences.
- Scalability; that is, it should be possible to accommodate a large number of processes, each with its own stream(s).
- Locality; that is, a process should be able to spawn a new sequence of random numbers without interprocess communication.

In this section we discuss four different techniques for transforming a sequential random number generator into a parallel random number generator.

10.3.1 Manager-Worker Method

Gropp et al. [45] have described a manager-worker approach to parallel random number generation. A "manager" process has the task of generating random numbers and distributing them to "worker" processes that consume them. Here are two disadvantages of the manager-worker approach.

Some random number generators produce sequences with long-range correlations. Because each process is sampling from the same sequence, there is a possibility that long-range correlations in the original sequence may become short-range correlations in the parallel sequences.

The manager-worker method is not scalable to an arbitrary number of processes. It may be difficult to balance the speed of the random number producer with the speed of the consumers of these numbers. It clearly does not exhibit locality. On the contrary, it is communication-intensive.

These disadvantages are significant, and this method is no longer popular. Let's consider methods in which each process generates its own random number sequence.

10.3.2 Leapfrog Method

The leapfrog method is analogous to a cyclic allocation of data to tasks. Suppose our parallel Monte Carlo method is executing on p processes. All processes use the same sequential random number generator. The process with rank r takes every pth element of the sequence, beginning with X_r:

$$X_r, X_{r+p}, X_{r+2p}, \ldots$$

Figure 10.5 illustrates the elements used by the process with rank 2 in a seven-process parallel execution in which each process generates its own random number sequence.

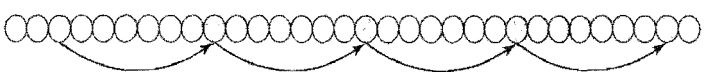


Figure 10.5 Process 2 (of 7) generates random numbers using the leapfrog technique.

It is easy to modify a linear congruential generator to incorporate leapfrogging. A jump of p elements is accomplished by replacing a with $a^p \bmod M$ and c with $c(a^p - 1)/(a - 1) \bmod M$. Makino has demonstrated how to modify lagged-Fibonacci generators to use leapfrogging [77].

Monte Carlo algorithms often require the generation of multidimensional random values. For instance, in the π estimation example we gave in Section 10.1, we generated coordinate pairs. If we want the parallel program to generate the same pairs as the sequential algorithm, the leapfrog method must be modified: we need to generate $(X_{2r}, X_{2r+1}, X_{2r+2p}, X_{2r+2p+1}, \ldots)$, not $(X_r, X_{r+p}, X_{r+2p}, X_{r+3p}, \ldots)$. This is a straightforward modification of the leapfrog method (Figure 10.6).

A disadvantage of the leapfrog method is that even if the elements of the original random number sequence have low correlation, the elements of the leapfrog subsequence may be correlated for certain values of p. This is especially likely to happen if p is a power of 2, a linear congruential generator is being used, and the modulus M is a power of 2. Even if this is not the case, leapfrogging can turn long-range correlations in the original sequence into short-range correlations in the parallel sequences.

Another disadvantage of the leapfrog method is that it does not support the dynamic creation of new random number streams.

10.3.3 Sequence Splitting

Sequence splitting is analogous to a block allocation of data to tasks. Suppose a random number generator has period P. The first P numbers emitted by the generator is divided into equal-sized pieces, one per process (Figure 10.7).

This method has the disadvantage of forcing each process to move ahead to its starting point in the sequence. This may take a long time. On the other hand, this only needs to be done at the initialization of the algorithm. After that, each process generates the elements in order.

Linear congruential generators with a power of 2 modulus have long-range correlations. Since the sequences produced by different processes represent elements far apart in the cycle, there may be correlations between the sequences produced by difference processes.

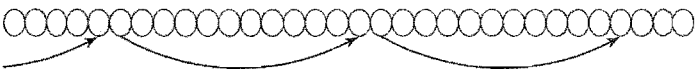


Figure 10.6 Process 2 (of 6) generates random number pairs in a modified leapfrog scheme.

Figure 10.7 In sequence splitting, each process is allocated a contiguous group of random numbers.

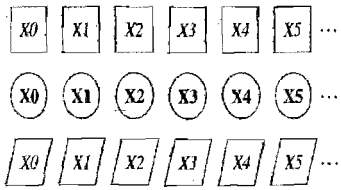


Figure 10.8 By using different parameters to initialize a sequential random number generator, it is often possible for each process to produce its own sequence.

Sequence splitting could be modified to support the dynamic creation of new sequences. For example, a process creating a new stream could give up half of its section to the new stream.

10.3.4 Parameterization

A fourth way to implement a parallel random number generator is to run a sequential random number generator on each process, but to ensure that each generator produces a different random number sequence by initializing it with different parameters (Figure 10.8).

Linear congruential generators with different additive constants produce different streams. Percus and Kalos have published a methodology for choosing the additive constant that works well for up to 100 streams [91].

Lagged Fibonacci generators are especially well suited for this approach. Providing each process with a different initial table of lag values allows each process to generate a different random number sequence. Obviously, correlations within lag tables or between lag tables would be fatal. One way to initialize the tables is to use a *different* lagged Fibonacci generator to generate the needed seed values. The processes could use the leapfrog technique or sequence splitting to ensure that they filled their tables with different values.

The number of distinct streams a lagged Fibonacci generator can produce is truly awesome [82]. For example, the default multiplicative lagged Fibonacci generator provided by the SPRNG library has around 2^{1008} distinct streams, allowing plenty of opportunities for creating new streams during the execution of the parallel program [83].

10.4 OTHER RANDOM NUMBER DISTRIBUTIONS

Our discussion to this point has focused on the problem of generating random numbers from a uniform probability density function. Sometimes we need to generate random numbers from other distributions.

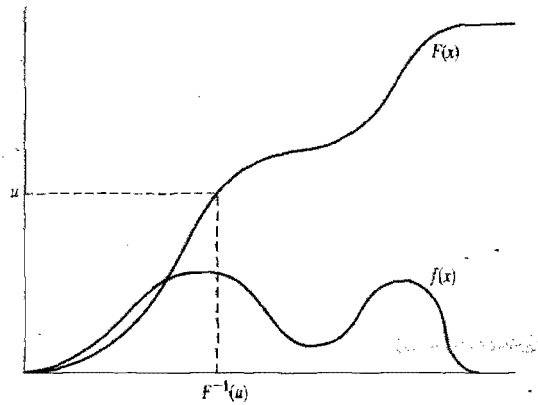


Figure 10.9 Given a probability density function $f(x)$, its cumulative distribution function $F(x)$, and u, a sample from a uniform distribution, then $F^{-1}(u)$ is a sample from $f(x)$.

10.4.1 Inverse Cumulative Distribution Function Transformation

Let u represent a sample from the uniform distribution $[0, 1)$.

Suppose we want to produce random variables from a probability density function $f(x)$. If we can determine the cumulative distribution function $F(x)$ and invert it, then $F^{-1}(u)$ is a random variable from the probability density function $f(x)$ (see Figure 10.9).

As an example of this transformation, we will derive a formula that yields a sample from the exponential distribution.

Exponential Distribution The decay of radioactive atoms, the distance a neutron travels in a solid before interacting with an atom, and the time before the next customer arrives at a service center are examples of random variables that are often modeled by an exponential probability density function.

The exponential probability density function with expected value m is the function $f(x) = (1/m)e^{-x/m}$. We can integrate $f(x)$ to find the cumulative distribution function $F(x) = 1 - e^{-mx}$. Inverting $F(x)$, we find the inverse function to be $F^{-1}(u) = -m \ln(1 - u)$. Since u is uniformly distributed between 0 and 1, there is no difference between u and $1 - u$. Hence the function $F^{-1}(u) = -m \ln u$ is exponentially distributed with mean m.

EXAMPLE 1

Produce four samples from an exponential distribution with mean 3.

■ Solution

We start with four samples from a uniform distribution:

$$0.540 \quad 0.619 \quad 0.462 \quad 0.095$$

Taking the natural logarithm of each value and multiplying by −3:

$$-3\ln(0.540) \quad -3\ln(0.619) \quad -3\ln(0.462) \quad -3\ln(0.095)$$

yields four samples from an exponential distribution with mean 3:

$$1.850 \quad 1.440 \quad 2.317 \quad 7.072$$

EXAMPLE 2

A simulation advances in time steps of 1 second. The probability of a particular event occurring is from an exponential distribution with mean 5 seconds. What is the probability of the event occurring in the next time step? How do we determine if the event happens in the next time step?

■ Solution

The probability of an event occurring in the next time step is 1/5. To determine if the event happens in the next time step, we generate a random number from the uniform distribution between 0 and 1. If the random number is less than or equal to 1/5, the event has occurred.

10.4.2 Box-Muller Transformation

We cannot invert the cumulative distribution function to come up with a formula yielding random numbers from the normal (gaussian) distribution

$$f(x) = \frac{1}{\sqrt{2\pi}} e^{-x^2/2}$$

Fortunately, the Box-Muller transformation allows us to produce a pair of standard deviates g_1 and g_2 from a pair of uniform deviates u_1 and u_2 [65]:

repeat
  $v_1 \leftarrow 2u_1 - 1$
  $v_2 \leftarrow 2u_2 - 1$
  $r \leftarrow v_1^2 + v_2^2$
until $r > 0$ and $r < 1$
$f \leftarrow \sqrt{-2\ln r / r}$
$g_1 \leftarrow f v_1$
$g_2 \leftarrow f v_2$

EXAMPLE 1

Produce four samples from a normal distribution with mean 0 and standard deviation 1.

u_1	u_2	v_1	v_2	r	f	g_1	g_2
0.234	0.784	−0.532	0.568	0.605	1.290	−0.686	0.732
0.824	0.039	0.648	−0.921	1.269			
0.430	0.176	−0.140	−0.648	0.439	1.935	−0.271	−1.254

Solution

From the uniform random samples 0.234 and 0.784 we derive the two normal samples −0.686 and 0.732. The next pair of uniform random samples 0.824 and 0.039 results in a value of $r > 1$, so we must discard these samples and generate another pair. The uniform samples 0.430 and 0.176 result in the normal samples −0.271 and −1.254.

EXAMPLE 2

Produce four samples from a normal distribution with mean 8 and standard deviation 2.

Solution

We modify the Box-Muller transformation by replacing the assignment

$$g_1 \leftarrow fv_1$$

with

$$g_1 \leftarrow 2fv_1 + 8$$

We do a similar replacement for the assignment to g_2.

u_1	u_2	v_1	v_2	r	f	g_1	g_2
0.017	0.262	−0.965	−0.475	1.158			
0.832	0.743	0.663	0.486	0.676	1.075	9.426	9.045
0.670	0.439	0.339	−0.122	0.130	5.602	11.800	6.630

The value of r resulting from the uniform samples 0.017 and 0.262 is too large, and we must reject these samples. However, the uniform samples 0.832 and 0.743 produce the normal samples 9.426 and 9.045. The uniform samples 0.670 and 0.439 produce the normal samples 11.800 and 6.630.

You can use the Box-Muller transformation to create a function that returns a single standard deviate. On the first, third, fifth, etc. invocations of this function, it performs the Box-Muller transformation, stores g_2, and returns g_1. On the second, fourth, sixth, etc. invocations of this function, the function returns the value of g_2 produced in the previous invocation.

10.4.3 The Rejection Method

The rejection method, first proposed by John von Neumann, allows us to produce samples from a probability density function $f(x)$ that we cannot integrate and/or invert analytically. Suppose we can generate samples for another probability density function $h(x)$, and we can find a constant δ such that $f(x) \leq \delta h(x)$ for all x (Figure 10.10). We produce samples from f in the following way: We generate a sample x_i from h and another sample u_i from the uniform distribution. If $u_i \delta h(x_i) \leq f(x_i)$ we accept x_i as a sample from $f(x)$ and return it. Otherwise, we repeat the test with another x_i and another u.

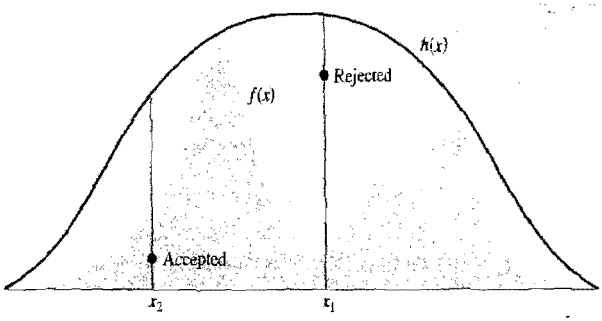


Figure 10.10 The rejection method allows us to generate samples from a probability density function $f(x)$. We produce x_i from probability density function $h(x)$ and u_i from the uniform probability density function over $(0, 1)$. In this figure $u_1 = 0.8$, $u_1\delta h(x_1) > f(x_1)$, and we reject sample x_1. On the other hand, $u_2 = 0.15$, $u_2\delta h(x_2) < f(x_2)$, and we accept sample x_2.

The points $(x_i, u_i\delta h(x_i))$ uniformly sample the area under the curve $\delta h(x)$. Since we only accept those points under the curve $f(x)$, the resulting sequence of x_is reflects the probability density function $f(x)$.

The rejection method works best when there is a relatively small amount of error between $f(x)$ and $\delta h(x)$. The larger this area, the greater the frequency at which candidate random numbers will be rejected, slowing the process. The efficiency of the rejection method can decrease sharply as the number of dimensions increases. For example, suppose that 75 percent of the random numbers are accepted for a one-dimensional integral. If the same efficiency holds true as the number of dimensions increases, the efficiency for a six-dimensional integral would be $(0.75)^6$, or about 18 percent.

EXAMPLE

A random variable has the probability density function

$$f(x) = \begin{cases} \sin x, & \text{if } 0 \le x \le \pi/4; \\ (-4x + \pi + 8)/(8\sqrt{2}), & \text{if } \pi/4 < x \le 2 + \pi/4; \\ 0, & \text{otherwise} \end{cases}$$

This probability density function is illustrated in Figure 10.11.

■ Solution

We can use the rejection method to generate random variables from this distribution. We need to find δ and $h(x)$ such that $f(x) \le \delta h(x)$ for all x. We note that the probability density function is greater than 0 for the values of x between 0 and $2+\pi/4$, and it has a maximum value of $\sqrt{2}/2$. We choose to use a uniform probability density function as our $h(x)$:

$$h(x) = \begin{cases} 1/(2 + \pi/4), & \text{if } 0 \le x \le 2 + \pi/4; \\ 0, & \text{otherwise} \end{cases}$$

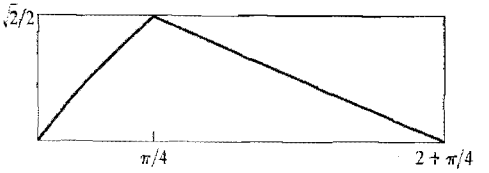


Figure 10.11 Using a uniform random variable and the rejection method to produce variables from a two-part probability density function.

If we multiply $h(x)$ by $\delta = (2 + \pi/4)(\sqrt{2}/2)$, then $\delta h(x) \geq f(x)$ for all x. Simplifying the terms, we see that

$$\delta h(x) = \begin{cases} \sqrt{2}/2, & \text{if } 0 \leq x \leq 2 + \pi/4; \\ 0, & \text{otherwise} \end{cases}$$

We generate a random number from the uniform distribution between 0 and 1 and multiply it by $2 + \pi/4$, giving us a random variable x_i from the uniform distribution between 0 and $2 + \pi/4$. Next we generate a random number u_i from the uniform distribution between 0 and 1. If $u_i \delta h(x_i) \leq f(x_i)$, then we accept x_i as a sample from $f(x)$ and return it. Otherwise we generate another pair (x_i, u_i) and repeat the test.

x_i	u_i	$u_i \delta h(x_i)$	$f(x_i)$	*Outcome*
0.860	0.975	0.689	0.681	Reject
1.518	0.357	0.252	0.448	Accept
0.357	0.920	0.650	0.349	Reject
1.306	0.272	0.192	0.523	Accept

10.5 CASE STUDIES

The five case studies in this section provide a glimpse into a few of the many domains in which Monte Carlo methods are useful.

10.5.1 Neutron Transport

We consider a simplified model of neutron transport in two dimensions (see Figure 10.12). A source emits neutrons against a homogeneous plate having thickness H and infinite height. A neutron may be reflected by the plate, absorbed by the plate, or it may pass through the plate. We wish to compute the frequency at which each of these events occurs as a function of plate thickness H.

Two constants that describe the interaction of the neutrons in the plate are the cross section of the capture C_c and the cross section of the scattering C_s. The total cross section $C = C_c + C_s$.

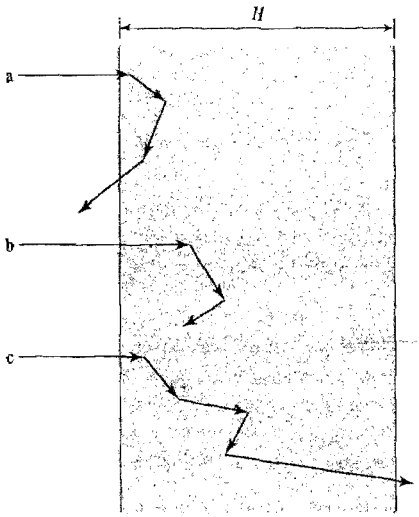


Figure 10.12 A neutron encountering a homogeneous medium may be (a) reflected, (b) absorbed, or (c) transmitted.

The distance L a neutron travels in the plate before interacting with an atom is modeled by an exponential distribution with mean $1/C$. As we saw in the previous section, if u is a random number from the uniform distribution $[0, 1)$, the formula

$$L = -\frac{1}{C} \ln u$$

is a random number from the appropriate exponential probability density function.

When a neutron interacts with an atom in the plate, the probability of bouncing off the atom is C_s/C, while the probability of being absorbed by the atom is C_c/C. We may use a random number from the uniform distribution $[0, 1)$ to determine the outcome of a neutron-atom interaction.

If a neutron scatters, it has an equal probability of moving in any direction. Hence its new direction D (measured in radians) can be modeled by a random variable uniformly distributed between 0 and π. (Since the plate has infinite height, we do not need to distinguish between bouncing upward and bouncing downward.) Given direction D, the actual distance in the x direction the neutron travels in the plate between collisions is $L \cos D$.

The simulation of a neutron continues until one of the following events occurs:

1. The neutron is absorbed by an atom.
2. The x position of the neutron is less than 0, meaning the neutron has been reflected by the plate.
3. The x position of the neutron is greater than H, meaning the neutron has been transmitted through the plate.

```
Neutron Transport Simulation:
  C — Mean distance between neutron/atom interactions is 1/C
  C_s — Scattering component of C
  C_c — Absorbing component of C
  H — Thickness of plate
  L — Distance neutron travels before collision
  d — Direction of neutron (measured in radians between 0 and π)
  u — Uniform random number
  x — Position of particle in plate (0 ≤ x < H)
  n — Number of samples
  a — True while particle still bouncing
  r, b, t — Counts of reflected, absorbed, transmitted neutrons
begin
  r, b, t ← 0
  for i ← 1 to n do
    d ← 0
    x ← 0
    a ← true
    while a do
      L ← −(1/C) × ln u
      x ← x + L × cos(d)
      if x < 0 then { Reflected }
        r ← r + 1
        a ← false
      else if x ≥ H then { Transmitted }
        t ← t + 1
        a ← false
      else if u < C_c/C then { Absorbed }
        b ← b + 1
        a ← false
      else
        d ← u × π
      endif
    endwhile
  endfor
  print r/n, a/n, t/n
end
```

Figure 10.13 Pseudocode for a neutron transport simulation using the Monte Carlo method.

Pseudocode for the neutron transport simulation appears in Figure 10.13. Note that time does not advance by the same amount in each iteration of the while loop. Instead, the simulation advances from one event (one interaction) to the next. This pseudo-time progression is called **Monte Carlo time.**

10.5.2 Temperature at a Point Inside a 2-D Plate

Imagine a very thin plate of homogeneous material. We wish to compute the steady-state temperature at a particular point in the plate. The top and the bottom of the plate are insulated, and the temperature at any point is solely determined by the temperatures surrounding it, except for the temperatures at the edges of the plate, which are fixed.

The interior temperature distribution is described by Laplace's equation, $\nabla^2 T = 0$, which means the temperature at a point is the average of the temperatures around it.

One approach to solving Laplace's equation numerically is to make the problem discrete by overlaying the plate with a two-dimensional mesh of points. In this case the temperature at a point is the average of the temperatures of the points above it, below it, to its right, and to its left (which we can think of as "north," "south," "east," and "west").

We can use a Monte Carlo technique to find the temperature at a particular point S. We compute the temperature of S by randomly choosing one of the four neighbors and adding its temperature to an accumulator. After we have sampled a random neighbor's temperature n times, we divide the sum by n to yield the temperature of S. This average has an expected value of $(T_n + T_s + T_e + T_w)/4$.

Of course, we do not know the temperatures of the neighboring points, but we could use the same technique to find their temperatures, too. Applying this idea recursively, we end up doing a **random walk** on the plate. The recursion and the random walk do terminate, because the temperatures on the edges of the plate are known.

Following are the Monte Carlo algorithm results (see Figure 10.14). We start at intersection S and randomly choose which direction to move (north,

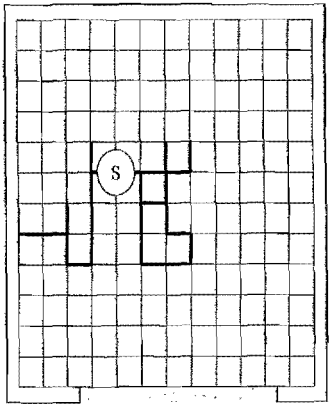


Figure 10.14 Use of a random walk to estimate the temperature of point S on a thin plate. The boundary temperatures are fixed. Edge points contacting the U-shaped white bar have temperature 0. Edge points contacting the gray bar have temperature 100. This random walk from S, illustrated by heavy lines, results in the temperature 0 being added to the sample.

south, east, or west). We continue to move in a random fashion until we hit one of the edges of the plate. At this point we add the temperature at the edge to our accumulator and repeat. At each iteration we can also determine the average edge temperature encountered over all random walks we have taken so far. We terminate the algorithm when the average temperature value converges.

10.5.3 Two-Dimensional Ising Model

The two-dimensional Ising model may be used to simulate the behavior of simple magnets as well as other phenomena (see Figure 10.15). The problem domain is a square lattice. Each intersection is called a **site.** Every site σ_k has an associated spin. Each spin can be in one of two states: up or down. We associate the value $1/2$ with up and the value $-1/2$ with down. The energy of the system is determined

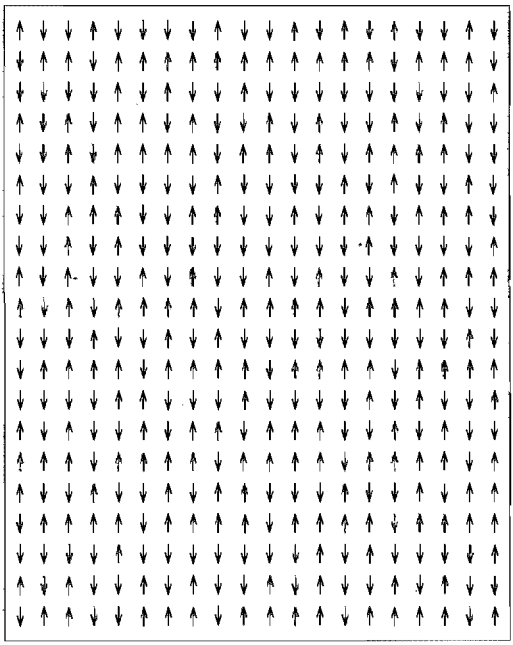


Figure 10.15 A 20 x 20 Ising model. Each of the 400 sites has an associated spin, either up or down. The energy of the system is a function of the spins. The model may take on any of 2^{400} different states. The probability of entering each of these states is influenced by both the current state of the system and its temperature.

by the function

$$E(\sigma) = -\sum_{i,j} J\sigma_i\sigma_j - B\sum_i \sigma_i$$

where the first sum is over nearest neighbors, J is a constant indicating the strength of the spin-spin interaction, and B is another constant having to do with the external magnetic field.

Our goal is to estimate the specific heat per particle, a problem analogous to performing integration over the possible configurations of the system. Given the temperature T and Boltzmann's constant k, the probability density function over all possible configurations is

$$\mu(\sigma) = \frac{e^{-E(\sigma)/kT}}{Z(T)}$$

where $Z(T)$ is a weighted sum over all states.

Unfortunately, it is difficult to sample from distribution μ. Here, a random sample x_i represents a configuration of spins. Note that the number of configurations is exceedingly large, even for small lattices. For example, our example 20×20 lattice has 400 sites. Each site has two possible values, meaning the number of configurations is 2^{400}. Because the probability density function is an inverse exponential function, the probabilities associated with most states are extremely small. If we try to take a uniform sample of the configurations, it is unlikely we will "hit" on enough of the higher-probability configurations to yield a good estimate of the integral. Instead, we need to find a sampling of the configurations that is biased toward those that have higher probability. The Metropolis algorithm generates such a sampling.

The **Metropolis algorithm** uses the current configuration x_i (current random sample) to generate the next configuration x_{i+1} (next random sample). Given x_i, the algorithm generates a neighboring configuration x'. If $E(x') < E(x_i)$, then $x_{i+1} = x'$. If $E(x') > E(x_i)$, then $x_{i+1} = x'$ with probability $e^{-[E(x')-E(x_i)]/kT}$; otherwise $x_{i+1} = x_i$. The series of random samples, called a **Markov chain,** represents a random walk through the possible configurations. When applied to the Ising model, the Metropolis algorithm takes the form shown in Figure 10.16.

While short series of random samples produced by the Metropolis algorithm are highly correlated, if the algorithm is allowed to produce enough samples, it can provide good coverage of an entire probability density function.

How can we be sure that the Markov chain of configurations visited by the Metropolis algorithm corresponds to the underlying probability density function? One way to be sure is to satisfy the detailed balance condition. Let $P(x_i)$ represent the probability of being in configurations x_i, and let $P(x_j \mid x_i)$ represent the probability of the random walk moving to configuration x_j from configuration x_i. The **detailed balance condition** holds if

$$P(x_1)P(x_2 \mid x_1) = P(x_2)P(x_1 \mid x_2)$$

```
Metropolis Algorithm:
  k — Boltzmann's constant
  T — Temperature
  E — Energy function
  Δ — Change in energy
  ρ — Probability of changing to state x'
  u — Uniform random variable

begin
  x_0 ← Initial state of model
  i ← 0
  repeat
    σ ← randomly selected site (from uniform distribution)
    x' ← Identical to x_i except spin at σ is reversed
    Δ ← E(x') − E(x_i)
    if Δ < 0 then
      ρ ← 1
    else
      ρ ← e^{−Δ/kT}
    endif
    if u < ρ then
      x_{i+1} ← x'
    else
      x_{i+1} ← x_i
    endif
    i ← i + 1
  forever
end
```

Figure 10.16 The Metropolis algorithm applied to the Ising model.

Suppose $E(x') > E(x_i)$. The Metropolis algorithm satisfies the detailed balance condition if

$$P(x_i)P(x' \mid x_i) = P(x')P(x_i \mid x')$$

$$\Rightarrow \frac{e^{-E(x_i)/kT}}{Z(T)} \times e^{-[E(x')-E(x_i)]/kT} = \frac{e^{-E(x')/kT}}{Z(T)} \times 1$$

$$\Rightarrow \frac{e^{-E(x')/kT}}{Z(T)} = \frac{e^{-E(x')/kT}}{Z(T)}$$

The equality also holds if $E(x') \leq E(x_i)$. Hence the Metropolis algorithm satisfies the detailed balance condition.

10.5.4 Room Assignment Problem

Given n, an even number of college freshmen, our goal is to assign them to $n/2$ rooms in a residence hall so that interpersonal conflicts are minimized. Every student has completed a survey, and a computer program has produced a table of "dislikes"—in other words, the value of entry (i, j) of the table indicates the extent to which students i and j are likely to get on each other's nerves. (The value

of entry $[i, j]$ equals the value of entry $[j, i]$.) We will solve this problem using a technique called simulated annealing.

Physical annealing is the process of heating a solid until it melts, then cooling it slowly. The purpose of physical annealing is to produce a strong, defect-free crystal with a regular structure. When the material is hot, the atoms are in a higher-energy state and more easily rearrange themselves. As the temperature drops, the atomic energies decrease, and the atoms do not rearrange themselves as easily. Slow cooling allows the material to reach a state of minimum energy, which is its crystalline form.

Simulated annealing makes an analogy between physical annealing and solving a combinatorial optimization problem. A solution to the optimization problem corresponds to a state of the material, the value of the objective function for a particular solution corresponds to the energy associated with a particular state, and the optimal solution to the problem corresponds to the minimum energy state.

Simulated annealing is an iterative algorithm. During each iteration the current solution is randomly changed to create an alternate solution in the neighborhood of the current solution. If the value of the objective function for the new solution is less than the value of the objective function for the current solution, then the new solution becomes the current solution. If the value of the objective function for the new solution is greater than the value of the objective function for the current solution, then the new solution becomes the current solution with probability $e^{-\Delta/T}$, where Δ is the difference between the values of the objective function and T is the current "temperature."

Why would we want to move to a solution that is inferior to one we have already found? The reason is that solution spaces usually have local minima. We do not want the algorithm to settle too quickly into a local minimum. When the temperature is higher, the algorithm can easily "climb out of" local minima (Figure 10.17a). When the temperature decreases, the probability of doing so is reduced (Figure 10.17b).

Note that simulated annealing and the Metropolis algorithm are closely related. Both use the same probability function to determine if a jump should be made to a higher-energy state. The difference is that in simulated annealing we are searching for the minimum value of the function, rather than computing an integral.

In order to solve a problem using simulated annealing, we must:

- decide how to represent solutions
- define the cost function
- define how to generate a new, neighboring solution from an existing solution
- design a cooling function

Let's go through each of these steps for the room assignment problem. We start with an incompatibility matrix D; entry $d_{i,j}$ is a floating-point value between 0 and 10 that indicates how much students i and j are going to dislike each other. Note that $d_{i,j} = d_{j,i}$.

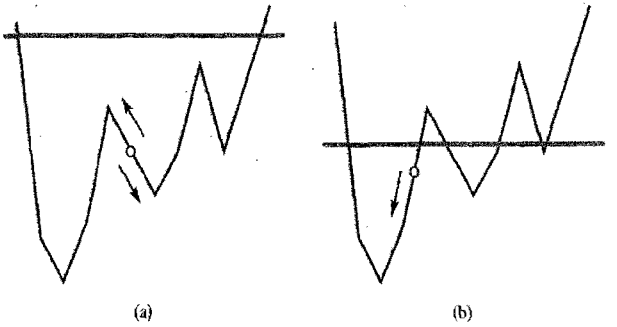


Figure 10.17 Simulated annealing always allows the search to move to a newly generated solution of lower cost. The probability of moving to a newly generated solution of higher cost shrinks as the temperature drops. (a) When the temperature is high, moving to a solution of higher cost is more probable. (b) When the temperature is low, moving to a solution of higher cost is less probable.

A solution is an assignment of the n students to $n/2$ rooms. We create array a to keep track of these assignments. Each entry a_i is an integer between 0 and $n/2 - 1$, representing the room person i is assigned to. Each value j in the range 0 through $n/2 - 1$ appears exactly twice in array a.

The cost function is simply the sums of the incompatibilities of the students in the rooms. Let r_i represent the roommate of student i. Then the cost function is defined to be

$$\sum_{i=0}^{n-1} d_{i,r_i}$$

We can generate a new solution near the current solution by choosing two students at random and switching their room assignments.

Finally, we need to choose the temperature function. The choice of temperature function can have a great effect on the performance of the algorithm. A poor function may cause a simulated annealing algorithm to find a poor solution, take too long to execute, or both.

For this problem we choose a simple geometric temperature function:

$$T_0 = 1$$

$$T_{i+1} = 0.999T_i$$

Figure 10.18 illustrates the convergence of the simulated annealing algorithm solving the room assignment problem using a geometric temperature function. Both algorithms find the same solution, but the algorithm starting with $T_0 = 10$ iterates twice as long as the algorithm starting with $T_0 = 1$.

Pseudocode for a simulated annealing solution to the room assignment problem appears in Figure 10.19.

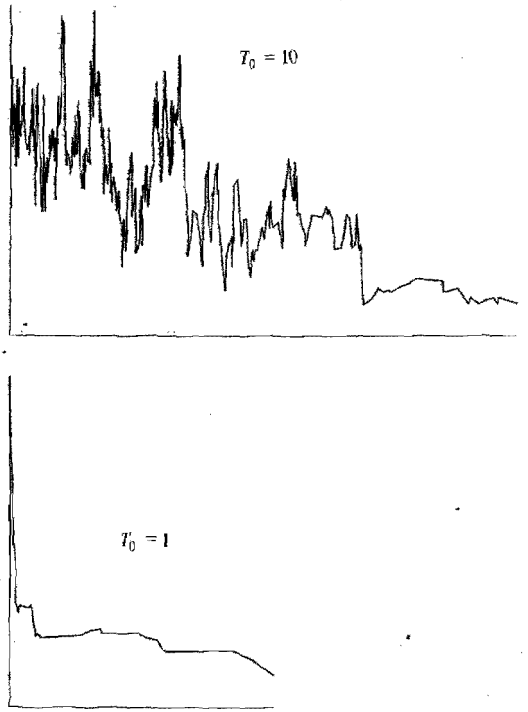


Figure 10.18 Convergence of simulated annealing algorithms solving the room assignment problem. In both cases the geometric temperature function $T_{i+1} = 0.999T_i$ is used. Both algorithms converge on the optimal solution. However, when the initial temperature is higher, the convergence is slower.

Simulated annealing is not guaranteed to find an optimal solution. In fact, the same algorithm using different streams of random numbers may converge on different solutions. Hence it makes sense to execute the same algorithm multiple times with different random number seeds. This is an obvious opportunity to use a parallel computer to speed overall execution time.

10.5.5 Parking Garage

A parking garage has S stalls. The length of time between successive arrivals of cars at the entrance to the garage is a random variable from a Poisson distribution with mean A minutes. If a car arrives at the garage and a stall is available, it occupies one of the stalls. The length of time a car stays in the garage is a random variable from a normal distribution with mean M minutes and standard deviation $M/4$ minutes. If a car arrives at the entrance and no stalls are available, the car is turned away. We wish to determine the steady-state characteristics of the parking

Room Assignment Problem:

```
a[0..n − 1] — n-element array containing room assignments
c1, c2 — two persons involved in possible room swap
d[0..n − 1, 0..n − 1] — n × n matrix containing roommate incompatibilities
sum — sum of dislikes of best solution found so far
new_sum — sum of dislikes of newly generated solution
t — temperature

begin
  Randomly assign students to rooms
  sum ← 0
  for i ← 0 to n − 1 do
    for j ← 0 to n − 1 do
      if a[i] = a[j] then
        sum ← sum + d[i][j]
    endfor
  endfor
  t ← 1
  i ← 0
  while i < 1000 do {Stop if no changes for 1000 iterations}
    repeat
      c1 ← ⌊u × n⌋
      c2 ← ⌊u × n⌋
    until a[c1] ≠ a[c2]
    Compute new_sum assuming c1 and c2 swap rooms
    if new_sum < sum or u ≤ e^((sum−new_sum)/t) then
      Swap room assignments for c1 and c2
      sum ← new_sum
      i ← 0
    else i ← i + 1
    endif
    t ← 0.999 × t
  endwhile
  print a and sum
end
```

Figure 10.19 Solving the room assignment problem using simulated annealing.

garage: the average number of stalls occupied by cars and the probability of a car being turned away because the garage is full.

We model time in minutes as a real variable t. When the simulation begins, $t = 0$.

We model the parking garage stalls as an array G with S elements. Element G_i contains the time that stall i is available. At the beginning of the simulation $G_i = 0$, for all i, $0 \leq i < S$.

We begin the simulation with the arrival of the first car; that is, it arrives at time 0.

Since car arrivals are characterized by a Poisson distribution, the time between car arrivals is an exponential distribution with mean A. As we saw in the previous section, we can use the expression $-A \ln u$ to determine the next car arrival time, where u is a random number uniformly distributed in $[0, 1)$.

We increment t by this amount and look for an available stall; that is, a stall i such that $G_i \leq t$.

When we assign a car to a stall i, we must reset G_i to reflect the time the car leaves the parking garage. Since this is a normal distribution, we can use the Box-Muller transformation described in Section 10.4.

10.5.6 Traffic Circle

A traffic circle (also called a rotary or a roundabout) is a way of handling traffic at an intersection without using signal lights. Often seen in Europe and the northeastern United States, traffic circles support the concurrent movement of multiple cars in the same direction.

Figure 10.20 illustrates a simple traffic circle. Traffic feeds into the circle from four roads, labeled N, W, S, and E. Every vehicle moves around the circle in a counterclockwise direction.

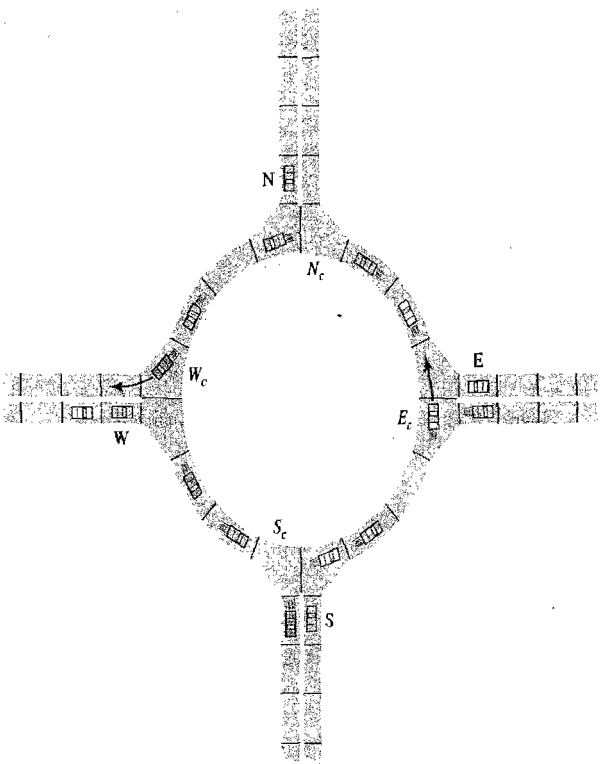


Figure 10.20 A model of a traffic circle. Cars within the traffic circle travel in a counterclockwise direction and take priority over cars trying to enter the circle.

	f
N	3
W	3
S	4
E	2

D	N	W	S	E
N	0.1	0.2	0.5	0.2
W	0.2	0.1	0.3	0.4
S	0.5	0.1	0.1	0.3
E	0.3	0.4	0.2	0.1

Figure 10.21 Probabilities associated with the traffic circle problem. Array element f_i is the mean time between vehicle arrivals at entrance i. Matrix element $d_{i,j}$ is the probability that a car entering the circle at i will exit at j.

In our simulation of the circle we divide the circle into 16 sections. During a single time step all vehicles inside the circle move to the next section in the counterclockwise direction (or leave the circle at one of the four exits). Vehicles within the traffic circle take priority over vehicles trying to enter the circle. Hence vehicles inside the circle are never prevented from moving forward during a time step.

A vehicle wishing to enter the circle may do so if there is no vehicle already in the circle attempting to enter the same zone. In Figure 10.20, for example, the vehicles waiting at N and S may enter the circle at the next time step, since there are no vehicles in the potential conflict zones S_c and N_c. The vehicle waiting at W may also enter the circle, because the car at W_c is leaving the circle. However, the vehicle waiting at E may not enter the circle, because the car at E_c is staying in the circle and has precedence.

To complete our model of the traffic circle, we must know the frequency at which cars arrive at the four access points. We must also know the frequency at which cars entering at a certain point exit at each of the four points. See Figure 10.21. The probability of a car arriving at an entrance during a particular time step is a random variable from an exponential distribution with mean m. Array f provides the mean time between arrivals at each of the four entrances. Element $d_{i,j}$ of matrix D is the probability that a car entering at i will exit at j. For example, the probability that a car entering at E will exit at S is 0.20.

Our goal is to construct a simulation of the traffic circle in order to answer two questions:

1. For each of the four traffic circle entrances, what is the probability that a car will have to wait before entering the circle?
2. For each of the four traffic circle entrances, what is the average length of the queue of vehicles waiting to enter the traffic circle?

Eight principal arrays are sufficient to perform the simulation and store the information needed to answer these two questions. See Figure 10.22. The traffic

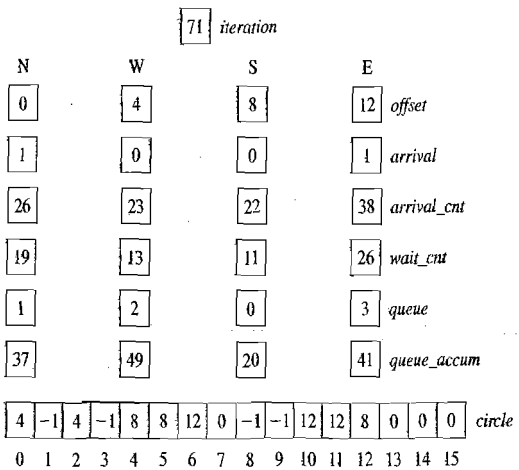


Figure 10.22 Data structures supporting the traffic circle simulation.

circle itself is represented by *circle*, a circular buffer implemented as an array of 16 integers. Array *offset* indicates the index in *circle* associated with each of the four entrances and exits. Index 0 represents the northern entrance/exit, index 4 is the location of the western entrance/exit, and so on. Each element of array *circle* represents a circle segment that is either empty or holds one car. If $circle[i] = -1$, the segment is empty. Otherwise, $circle[i]$ contains an integer that represents the car's exit (0, 4, 8, or 12).

When cars arrive at one of the entrances to the traffic circle, the appropriate element of array *arrival* is set to 1 at that time step. Array *arrival_cnt* contains the total number of arrivals at each entrance, and array *wait_cnt* is a count of the number of cars that could not enter the traffic circle immediately. Array *queue* keeps track of how many cars are waiting to enter the traffic circle at each entrance, and array *queue_accum* is a total, over all time steps of the simulation, of the values in *queue*.

Pseudocode for the traffic circle simulation appears in Figure 10.23. Each time step of the simulation is divided into three phases. First, new cars arrive at the traffic circle. Second, cars already inside the traffic circle move forward. (Array *new_circle*, not shown in Figure 10.22, facilitates this phase.) Cars that reach their destination exit are removed from the circle. Note that there is no need to simulate the lanes leading away from the circle. Third, cars enter the circle if there is room.

When a car does enter the traffic circle, the simulation must determine the desired exit of that car by generating a uniform random variable and referring to matrix D. In the pseudocode this step is represented by a call to function *ChooseExit*. We illustrate this process with an example. Suppose a car is entering

Traffic Circle Simulation:

```
Data Structures Representing the Traffic Circle
  circle[0..15] — Current state of traffic circle
  new_circle[0..15] — Next state of traffic circle

Data Structures Representing the Four Entrances
  offset[0..3] — Each entrance's location (index) in traffic circle
  arrival[0..3] — 1 if a car arrived this time step
  wait_cnt[0..3] — Number of cars that have had to wait
  arrival_cnt[0..3] — Total number of cars that have arrived
  queue[0..3] — Number of cars waiting to enter circle
  queue_accum[0..3] — Accumulated queue size over all time steps

begin
 for i ← 0 to 15 do
  circle[i] ← −1
 endfor
 for i ← 0 to 3 do
  arrival_cnt[i], wait_cnt[i], queue[i], queue_accum[i] ← 0
 endfor
 for iteration ← 0 to requested_iterations
  { New cars arrive at entrances }
  for i ← 0 to 3 do
   if u ≤ 1/f[i] then { u is a uniform random number }
    arrival[i] ← 1
    arrival_cnt[i] ← arrival_cnt[i] +1
   else arrival[i] ← 0
   endif
  endfor
  { Cars inside circle advance simultaneously }
  for i ← 0 to 15 do
   j ← (i +1) mod16
   if circle[i] = −1 or circle[i] = j then new_circle[j] ← −1
   else new_circle[j] ← circle[i]
   endif
  endfor
  circle ← new_circle
  { Cars enter circle }
  for i ← 0 to 3 do
   if circle[offset[i]] = −1 then
    {There is space for car to enter}
    if queue[i] > 0 then
     {Car waiting in queue enters circle}
     queue[i] ← queue[i] −1
     circle[offset[i]] ← Choose_Exit(i)
    else if arrival[i] > 0
     {Newly arrived car enters circle}
     arrival[i] ← 0
     circle[offset[i]] ← Choose_Exit(i)
    endif
   endif
```

Figure 10.23 Pseudocode for traffic circle simulation.

```
      if arrival[i] > 0 then
        {Newly arrived car queues up}
        wait_cnt[i] ← wait_cnt[i] + 1
        queue[i] ← queue[i] + 1
      endif
    endfor
    for i ← 0 to 15 do
      queue_accum[i] ← queue_accum[i] + queue[i]
    endfor
  endfor {iteration}
end
```

Figure 10.23 (contd.) Pseudocode for traffic circle simulation.

from the west, and we generate the random variable 0.55. We work through row W of matrix D until the total of the probabilities exceeds 0.55. The first entry is 0.2, which is not greater than 0.55. That means the destination is not the north exit. The second entry is 0.1. Adding this value to the first gives us 0.3. Since 0.3 is not greater than 0.55, the destination is not the west exit. Adding the third entry, 0.3, to the total gives us 0.6. Since 0.6 is greater than 0.55, the south exit is the destination. The car is entering at the west entrance (offset 4) and leaving at the south exit (offset 8). Hence we perform the assignment *circle*[4] ← 8.

When the traffic circle simulation begins, there are no cars inside the traffic circle, and delays are at a minimum. As the simulation progresses, traffic jams develop and then dissipate. The simulation should continue until the answers to the two questions have converged.

10.6 SUMMARY

Monte Carlo methods use statistical sampling to find approximate solutions to a wide variety of problems. Two important applications of Monte Carlo methods are numerical integration and simulation. Monte Carlo methods are superior to deterministic numerical algorithms for finding integrals when the number of dimensions is larger than about six. It is difficult to derive analytical answers to many questions arising from systems with stochastic behavior. Monte Carlo simulations of these systems can be good tools for generating approximate answers to these questions.

In order to produce reliable results, a Monte Carlo method must have access to a good stream of random numbers. The maximum period of a random number generator returning 32-bit integers is 2^{32}, or about four billion. This is too small a period for modern computers. Make sure you use a generator that has at least 48 bits of precision.

Sometimes a random number generator that is good in general may not work well for a particular application. If you have a critical application, it is a good idea to run it twice using two different random number generators to see if both runs produce similar results.

A variety of methods have been proposed for generating random numbers on a parallel computer, including the leapfrog method, sequence splitting, and maintaining independent sequences.

The most popular random number generators produce a pseudo-random sequence of values from a uniform distribution. Often a Monte Carlo method requires a random number from another distribution. Straightforward algorithms exist to transform samples from a uniform distribution into samples from an exponential distribution or a normal (gaussian) distribution. The rejection method allows us to produce numbers from other distributions.

We have considered six applications of the Monte Carlo method that demonstrated a variety of solution techniques. In the process of solving these problems we introduced two important algorithms. The Metropolis algorithm is a particularly good way to produce a sample from a high-dimensional space. Simulated annealing is an algorithm for finding approximate solutions to combinatorial optimization problems.

10.7 KEY TERMS

detailed balance condition
linear congruential generator
Markov chain
Metropolis algorithm
Monte Carlo method
Monte Carlo time
multiplicative congruential generator
period
pseudo-random number generator
random walk
seed
simulated annealing
site
uniform distribution

10.8 BIBLIOGRAPHIC NOTES

Easy-to-understand introductions to the Monte Carlo method and the Metropolis algorithm appear in *Computational Physics: Problem Solving with Computers* by R. Landau and Paez [65]. In contrast, *A Guide to Monte Carlo Simulations in Statistical Physics* by D. Landau and Binder provides a more rigorous presentation of the design and implementation of Monte Carlo simulations and the analysis of their results [64]. I first saw the "raindrops in cake pans" analogy in another introductory book, *Monte Carlo Methods*, written by Kalos and Whitlock [58].

Lehmer published the linear congruential method in 1951 [69]. For a time it was called "Lehmer's algorithm." Work on linear congruential generators with much longer periods continues. Wu gives a multiplicative congruential generator with the large prime-modulus $2^{61} - 1$ and four forms of multipliers [118]. However, L'Ecuyer and Simard warn that this generator fails a test of independence between the number of 1s in the binary representations of consecutive random numbers [68].

A variety of algorithms have been proposed for generating random variables from important nonuniform distributions. Wallace describes a fast way to generate normal and exponential random variables without relying on a source of uniform random variables [110]. Leva presents a fast algorithm for generating normal random variables that requires on average only 0.012 logarithm evaluations per standard deviate [72]. Marsaglia and Tsang describe a fast method for generating normal, exponential, and other random variables [80].

Mascagni surveys methods for generating parallel streams of random numbers via parameterization rather than sequence splitting [84]. His article contains a useful bibliography of earlier work.

The Scalable Parallel Random Number Generators (SPRNG) library, briefly documented by Mascagni and Srinivasan in *ACM Transactions on Mathematical Software* [83], is freely available from Florida State University. The URL is `http://sprng.cs.fsu.edu`.

The traffic circle problem is based on an example from Manno's *Introduction to the Monte-Carlo Method* [78].

10.9 EXERCISES

10.1 Suppose you are using the Monte Carlo method to compute an integral. The methodology is similar to the π-finding example in Section 10.1, except that the function to be integrated has 20 dimensions rather than 2. What sort of problem should you look out for if you are using a linear congruential random number generator?

10.2 An approach to parallel random number generation not discussed in the book is to assign each process the same linear congruential generator (with identical values for the multiplier, additive constant, and modulus). However, each process starts with a different seed value X_0. What is the principal risk associated with this approach?

10.3 Write a C function that uses the Box-Muller transformation to return a double-precision floating-point number representing a random value from the normal distribution.

10.4 A cylindrical hole with diameter d is drilled completely through a cube with edge length s so that the center of the cylindrical hole intersects two opposite corners of the cube. (See Figure 10.24.) Write a program to determine, with five digits of precision, the volume of the portion of the cube that remains when $s = 2$ and $d = 0.3$. Hint: The distance between the point (x_1, y_1, z_1) and the line $x = y = z$ is

$$\sin\left[\cos^{-1}\left(\frac{x_1 + y_1 + z_1}{\sqrt{3}\sqrt{x_1^2 + y_1^2 + z_1^2}}\right)\right]\sqrt{x_1^2 + y_1^2 + z_1^2}$$

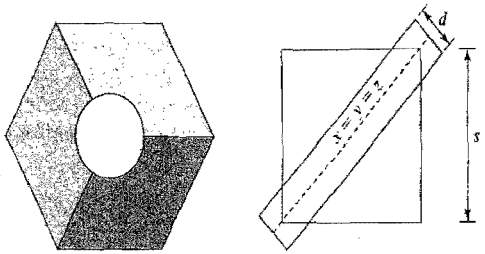


Figure 10.24 Two views of the object described in Exercise 10.4. (a) Looking down on one corner of the cube, we see that the hole goes completely through to the opposite corner. (b) Looking from the side of the cube, we see that all material within distance $d/2$ of the line $x = y = z$ is removed.

10.5 Write a program to evaluate the definite integral

$$\int_{x=0}^{4}\int_{y=0}^{3}\int_{z=0}^{2} 4x^3 + xy^2 + 5y + yz + 6z \, dz \, dy \, dx$$

to five digits of precision.

10.6 Write a program to evaluate the definite integral

$$\int_{x=0}^{4}\int_{y=0}^{3}\int_{z=0}^{x+y} 4x^3 + xy^2 + 5y + yz + 6z \, dz \, dy \, dx$$

to five digits of precision.

10.7 A radioactive atom has a mean lifetime of m time units. The probability that a radioactive atom will decay in any given time unit is $(1/m)e^{-1/m}$. Given an initial pool of 100,000 radioactive atoms, compute how many atoms decay at each time step in the first 1,000 time units, when $m = 250$.

10.8 Implement a parallel program solving the neutron transport problem described in Section 10.5.1. Let $C_c = 0.3$ and $C_s = 0.7$. Determine the probability of absorption, reflection, and transmission for $H = 1, 2, 3, \ldots, 10$. Base your results on 10 million tests (neutrons) for each value of H.

10.9 Implement a parallel program solving the steady-state temperature problem described in Section 10.5.2. Assume the square plate has been discretized into a 20 × 20 grid of smaller squares. Assume the temperature on three sides of the plate is 0° and the temperature on the fourth side is 100°. Compute the temperature at the middle of the plate to three digits of precision.

10.10 Write a parallel program implementing the Ising model described in Section 10.5.3. The objective is to find the energy level of a 100 x 100 system after 1,000,000 iterations. Let $J = 1$, $B = 0$, and $kT = 1$. Give the system a "cold start" by initializing every site σ_k to "up" in state x_0. Evaluate $\sum_{i,j} J\sigma_i\sigma_j$ for every pair of sites that are horizontally or vertically adjacent. Repeat the experiment 1,000 times.

10.11 Implement a parallel program solving the room assignment problem posed in Section 10.5.4. Assume $n = 20$ and $T = 1$. Use a random number generator to construct matrix D. Each entry should be a uniform random variable between 0 and 10. Each process should solve the problem for the same matrix D, but with different seeds for the random number generator.

10.12 Implement a parallel program solving the parking garage problem posed in Section 10.5.5. Assume $S = 80$, $A = 3$, and $M = 240$. Determine the average number of stalls occupied by cars, and the probability of a car being turned away because the garage is full, as $t \to \infty$, that is, in the steady state.

10.13 Implement a parallel program solving the traffic circle problem posed in Section 10.5.6. Use the program to answer these two questions:

a. For each of the four traffic circle entrances, what is the steady state probability that a car will have to wait before entering the circle?

b. For each of the four traffic circle entrances, what is the average length of the queue of vehicles waiting to enter the traffic circle, in the steady state?

CHAPTER

11

Matrix Multiplication

We go on multiplying our conveniences only to multiply our cares. We increase our possessions only to the enlargement of our anxieties.

Anna C. Brackett, *The Technique of Rest*

11.1 INTRODUCTION

Considering how often the matrix multiplication algorithm is presented in computer science classes, it's ironic that few scientific and engineering problems require the multiplication of large matrices. Here are two domains in which matrix multiplication is used. Computational chemists represent some problems in terms of states of a chemical system. Each index corresponds to a different basis state, and the matrix approximates the Hamiltonian of the system. A change of basis is accomplished through matrix multiplication. As another example, some transforms used in signal processing rely on the multiplication of large matrices.

This chapter presents two sequential matrix multiplication algorithms and then explores two different approaches to parallel matrix multiplication. In Section 11.2 we review the standard sequential matrix multiplication algorithm. Charting the algorithm's performance as matrix sizes increase, we see how performance drops dramatically once the second factor matrix no longer fits inside cache memory. We then show how a recursive implementation of matrix multiplication that multiplies blocks of the original matrices can maintain a high cache hit rate.

In Section 11.3 we design a parallel algorithm based upon a rowwise block-striped decomposition of the matrices. We derive an expression for the expected computation time of this algorithm, and we analyze its isoefficiency. In Section 11.4 we go through the same design and analysis methodology for a parallel algorithm based on a checkerboard block decomposition of the matrices.

11.2 SEQUENTIAL MATRIX MULTIPLICATION

11.2.1 Iterative, Row-Oriented Algorithm

The product of an $l \times m$ matrix A and an $m \times n$ matrix B is an $l \times n$ matrix C whose elements are defined by

$$c_{i,j} = \sum_{k=0}^{m-1} a_{i,k} b_{k,j}$$

A sequential algorithm implementing matrix multiplication appears in Figure 11.1. The algorithm requires lmn additions and the same number of multiplications. Hence the time complexity of multiplying two $n \times n$ matrices using this sequential algorithm is $\Theta(n^3)$. Sequential matrix multiplication algorithms with a lower time complexity have been developed, such as Strassen's algorithm, but every algorithm developed in this chapter is a parallelization of the straightforward algorithm.

It's easy to implement this algorithm. We've benchmarked a C implementation of this matrix multiplication algorithm on a node of a Beowulf cluster: a Linux computer with a 933 MHz Pentium III CPU with a 233 Kilobyte level 2 cache. The results of the benchmarking appear in Figure 11.2. For smaller matrices the execution speed is about 220 megaflops, but for larger matrices the execution speed is about 80 megaflops. What accounts for this drop in performance?

Consider Figure 11.3. During each iteration of the outer i loop, every element of matrix B is read. If matrix B is too large for the cache, then later elements read into cache displace earlier elements read into cache, meaning that in the next

Matrix Multiplication (row-oriented):

```
Input:
  a[0..l − 1, 0..m − 1]
  b[0..m − 1, 0..n − 1]

Output:
  c[0..l − 1, 0..n − 1]

for i ← 0 to l − 1
  for j ← 0 to n − 1
    c[i, j] ← 0
    for k ← 0 to m − 1
      c[i, j] ← c[i, j] + a[i, k] × b[k, j]
    endfor
  endfor
endfor
```

Figure 11.1 Iterative, row-oriented matrix multiplication algorithm.

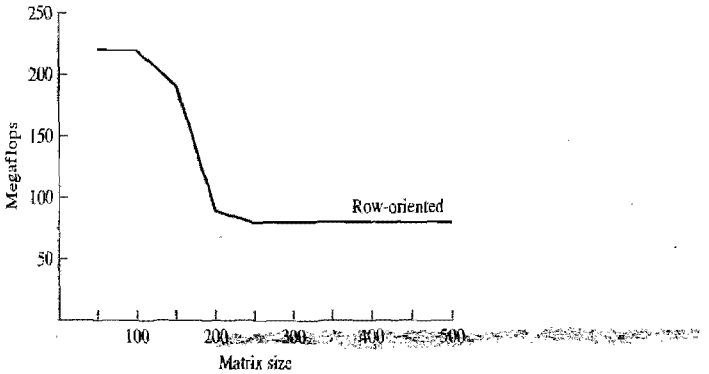


Figure 11.2 Performance of row-oriented matrix multiplication algorithm on a computer with a 933 MHz Pentium III CPU. When matrix B no longer fits in the cache, the performance of the row-oriented matrix multiplication algorithm drops sharply.

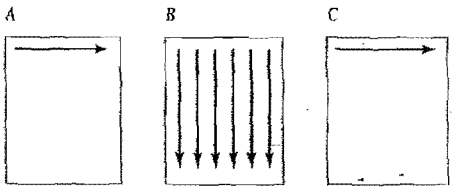


Figure 11.3 In a single iteration of the loop indexed by i, row i of matrix A and all of matrix B are read, while row i of C is written.

iteration of the loop indexed by i, all of the elements of B will need to be read into cache again. Hence once the matrices reach a certain size, the cache hit rate falls dramatically, lowering the performance of the CPU.

The CPU we used for benchmarking has a 256 Kilobyte cache. We are multiplying double-precision floating-point numbers, meaning each matrix element fills eight bytes. Hence the cache can hold at most 32,768 matrix elements. The square root of 32,768 is about 181. The performance of the algorithm reflects that when $n \leq 150$, the cache hit rate is much higher than when $n \geq 200$.

11.2.2 Recursive, Block-Oriented Algorithm

In order to perform the matrix multiplication AB, the number of columns of A must be equal to the number of rows of B.

```
double a[N][N], b[N][N], c[N][N];

void mm (int crow, int ccol, /* Corner of C block */
         int arow, int acol, /* Corner of A block */
         int brow, int bcol, /* Corner of B block */
         int l,              /* Block A is l x m */
         int m,              /* Block B is m x n */
         int n)              /* Block C is l x n */
{
   int lhalf[3], mhalf[3], nhalf[3]; /* Quadrant sizes */
   int i, j, k; double *aptr, *bptr, *cptr;

   if (m * n > THRESHOLD) {

      /* B doesn't fit in cache---multiply blocks of A, B */

      lhalf[0] = 0;  lhalf[1] = l/2;  lhalf[2] = l - l/2;
      mhalf[0] = 0;  mhalf[1] = m/2;  mhalf[2] = m - m/2;
      nhalf[0] = 0;  nhalf[1] = n/2;  nhalf[2] = n - n/2;
      for (i = 0; i < 2; i++)
         for (j = 0; j < 2; j++)
            for (k = 0; k < 2; k++)
               mm (crow+lhalf[i], ccol+mhalf[j],
                   arow+lhalf[i], acol+mhalf[k],
                   brow+mhalf[k], bcol+nhalf[j],
                   lhalf[i+1], mhalf[k+1], nhalf[j+1]);
   } else {

      /* B fits in cache --- do standard multiply */

      for (i = 0; i < l; i++)
         for (j = 0; j < n; j++) {
            cptr = &c[crow+i][ccol+j];
            aptr = &a[arow+i][acol];
            bptr = &b[brow][bcol+j];
            for (k = 0; k < m; k++) {
               *cptr += *(aptr++) * *bptr; bptr += N;
            }
         }
   }
}
```

Figure 11.4 C function implementing recursive, block-oriented matrix multiplication. The initial call to this function is `mm (0, 0, 0, 0, 0, 0, N, N, N)`.

Let's suppose A has l rows and m columns, while B has m rows and n columns. If we divide A into four smaller matrices

$$A = \begin{pmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{pmatrix}$$

and divide B into four smaller matrices

$$B = \begin{pmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{pmatrix}$$

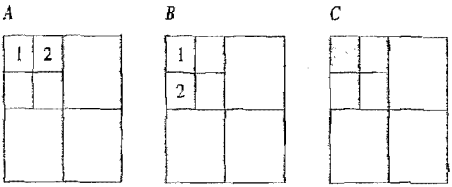


Figure 11.5 A recursive matrix multiplication algorithm breaks the matrices into smaller and smaller blocks until they can fit in cache. Here the algorithm has recursed twice before the blocks are small enough. Each block of C is the sum of the results of two block matrix multiplications.

such that the number of columns in A_{00} and A_{10} is equal to the number of rows in B_{00} and B_{01}, then the matrix product

$$C = \begin{pmatrix} A_{00}B_{00} + A_{01}B_{10} & A_{00}B_{01} + A_{01}B_{11} \\ A_{10}B_{00} + A_{11}B_{10} & A_{10}B_{11} + A_{11}B_{11} \end{pmatrix}$$

where each $A_{ik}B_{kj}$ represents multiplication of the block matrices and each + represents matrix addition.

Our goal is to compute the matrix product $C = AB$. If matrix B is too large to fit into cache, we can divide it into four pieces and use the idea of block matrix multiplication to compute C. If block B_{ij} is too large to fit into cache, we can apply this idea recursively until we have blocks that do fit in cache. A C implementation of the resulting recursive algorithm appears in Figure 11.4.

Figure 11.5 illustrates how the recursive matrix multiplication algorithm works. In this example, matrix B is too large for cache, so it is divided into four pieces. Each of the four pieces is still too large, so the algorithm recurses a second time.

We've benchmarked a C implementation of this recursive matrix multiplication algorithm on the same computer we used to measure the speed of the straightforward algorithm. The results of both benchmarking experiments appear in Figure 11.6. The recursive algorithm maintains high performance, even as the sizes of the matrices grow well beyond the cache capacity.

11.3 ROWWISE BLOCK-STRIPED PARALLEL ALGORITHM

In this section we develop a parallel matrix multiplication algorithm based upon a rowwise block-striped decomposition of the matrices.

11.3.1 Identifying Primitive Tasks

Each element of the product matrix C is a function of elements in A and B. Since A and B are not modified during the algorithm, it is possible to compute every

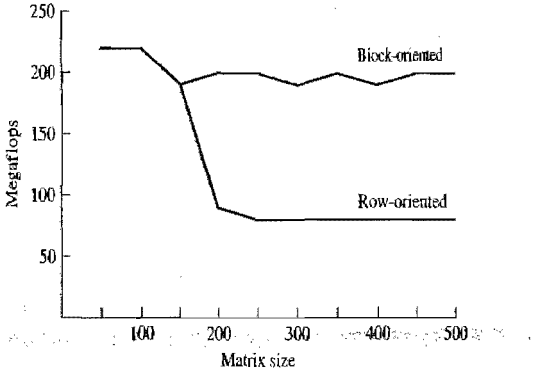


Figure 11.6 Performance of both sequential matrix multiplication algorithms on a computer with a 933 MHz Pentium III CPU. The block-oriented matrix multiplication algorithm keeps the cache hit rate high and achieves better performance than the row-oriented algorithm.

element of C simultaneously. As a first step in our parallel design, then, we can associate one primitive task with every element of C.

Precisely which elements does each of these tasks need? Computing element $c_{i,j}$ of the product matrix involves finding the inner product (dot product) of row i of A and column j of B.

11.3.2 Agglomeration

We can use this data dependence information to agglomerate tasks. It is natural to agglomerate tasks associated with either a row of C or a column of C, since they share a need for either a row of A or a column of B, respectively. Algorithms based on either of these design choices are quite similar. Let's choose to agglomerate tasks associated with a row of C.

It's simpler if we use the same agglomeration for all matrices. That way, the result of one matrix multiplication can be used as either factor matrix in another matrix multiplication. We assume, then, that each task is responsible for corresponding rows of A, B, and C.

Let's think about what task i can do with row i of A, row i of B, and row i of C. Recall that

$$c_{i,j} = \sum_{k=0}^{n-1} a_{i,k} b_{k,j}$$

With row i of matrices A and B, the task can compute $a_{i,i}b_{i,0}$, which is one of the terms of $c_{i,0}$. It can also compute $a_{i,i}b_{i,1}$, which is one of the terms of $c_{i,1}$, and so on. In other words, the task can perform n multiplications that represent partial sums for the n elements of row i of C.

Then what? We've already noted in an earlier section that row i of C is the product of row i of A and matrix B. So to complete its work each task must eventually access each row of B.

11.3.3 Communication and Further Agglomeration

If we organize the tasks as a ring, and each task passes its row of B to the next task in the ring, after a series of m iterations every task will have had possession of every row of B.

Figure 11.7 illustrates this process assuming there are four rows in C.

The number of processes on which we execute our parallel algorithm is probably much less than the number of rows in the product matrix, so we need to think of further agglomeration. Given the communication pattern we have developed, it makes sense to use a rowwise block-striped decomposition scheme.

11.3.4 Analysis

To simplify our analysis, we assume that A, B, and C are all $n \times n$ matrices. We also assume that n is a multiple of p, the number of active processes. Each process controls the same n/p rows of A and C throughout the algorithm. The contiguous groups of n/p rows of B are passed from process to process as illustrated in Figure 11.7.

When the algorithm begins, each process initializes its $(n/p) \times n$ portion of C to 0. During each iteration every process multiplies an $(n/p) \times n/p$ block of A by the $(n/p) \times n$ portion of B it currently possesses. It adds the resulting $(n/p) \times n$ matrix to its portion of C. If χ is the time needed for one of the add-multiply steps inside an inner product, the computational time of each iteration is

$$\chi(n/p)(n/p)n = \chi n^3/p^2$$

During every iteration, each process must also communicate its portion of B to the next process on the ring. If the communication is done after the computation, the time needed to send these elements would add $\lambda + (n/p)n/\beta$ to the execution time of each iteration. Receiving the next section of B from the predecessor ring process would occur at the same time.

The algorithm has p iterations. The total computation and communication time, then, is

$$p[\chi n^3/p^2 + \lambda + n^2/(p\beta)] = \chi n^3/p + p\lambda + n^2/\beta$$

Let's double-check on this expression. The sequential algorithm would have execution time χn^3. Since the computations are divided perfectly among the processes, it makes sense that the computational portion of the parallel algorithm has execution time $\chi n^3/p$.

Every process sends p messages, so the $p\lambda$ term also makes sense. Finally, every process handles all of B and sends all of it (one piece at a time) to its successor process, so the n^2/β term fits.

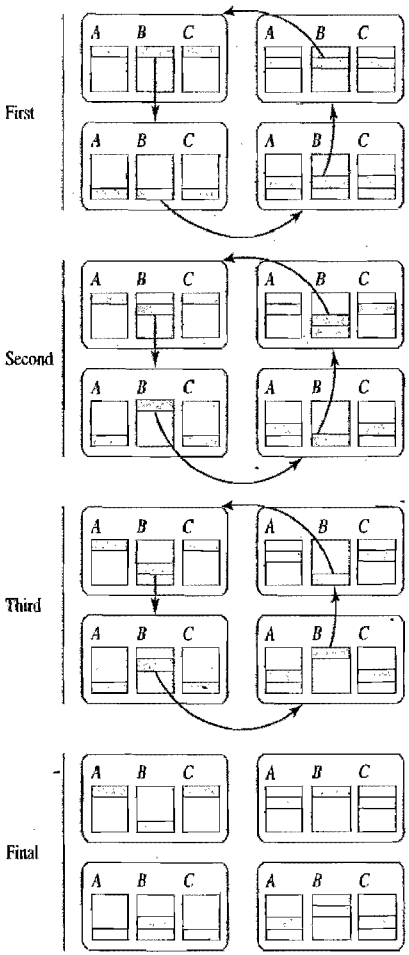


Figure 11.7 Communication of B in row-oriented parallel matrix multiplication algorithm. Each task is responsible for a row of A, a row of B, and a row of C. If B has m rows, then after $m-1$ communication steps each task has had access to every row of B.

Let's determine the isoefficiency of the rowwise block-striped matrix multiplication algorithm. The sequential algorithm has time complexity $\Theta(n^3)$. The communication complexity of the parallel algorithm is $\Theta(n^2)$. We multiply the community complexity by the number of processors p to get the overhead term: $T_o(n, p) = \Theta(pn^2)$. Hence the isoefficiency relation for the rowwise block-striped

matrix multiplication algorithm is

$$n^3 \geq Cpn^2 \Rightarrow n \geq Cp$$

We note that the memory utilization function $M(n) = n^2$. Let's determine how memory utilization per processor must increase in order to maintain a constant level of efficiency:

$$M(Cp)/p = C^2p^2/p = C^2p$$

This algorithm is not highly scalable.

Finally, this algorithm presents a good opportunity for overlapping communications with computation. Assuming there is enough memory to receive a new section of B while performing computations on the current section, each process could initiate its send/receive of B sections before it performed the matrix multiplication step. Since the communication complexity is $\Theta(n^2)$ and the computational complexity is $\Theta(n^3)$, the communication step can be almost completely overlapped with computations when the matrix sizes are large enough. (The time to initiate the communication cannot be overlapped with a computation.) When this happens, speedup can be very high.

Can we do better?

Let's consider the computation/communication ratio of the parallel row-oriented algorithm. When multiplying two $n \times n$ matrices on p processes, where n is a multiple of p, each process iterates through p iterations of a loop in which it multiplies an $(n/p) \times (n/p)$ submatrix of A with an $(n/p) \times n$ submatrix of B. Since the matrix multiplication steps are interleaved with communication steps in which elements of B are being passed from process to process, the ratio of computations per element of B is

$$\frac{2n^3/p^2}{n^2/p} = \frac{2n}{p}$$

The ratio is relatively low, because the submatrices of B have p times as many columns as rows.

In the next section we will develop an algorithm that improves the computation-to-communication ratio.

11.4 CANNON'S ALGORITHM

In this section we develop a parallel algorithm based upon a checkerboard block decomposition of the matrices. The algorithm is often referred to as **Cannon's algorithm** [14].

11.4.1 Agglomeration

The row-oriented parallel algorithm has a low ratio of computations per element of B because the blocks of B being manipulated are short and fat—having p times as many columns as rows.

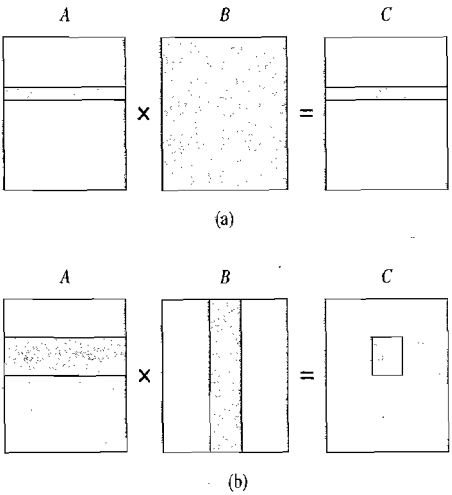


Figure 11.8 Comparison of number of elements of A and B needed to compute a process's portion of C in the two parallel matrix multiplication algorithms. (a) In the row-oriented algorithm, each process is responsible for computing n/p rows of C. It needs to reference n/p rows of A and every element of B. (b) In Cannon's algorithm, each process is responsible for computing an $(n/\sqrt{p}) \times (n/\sqrt{p})$ block of C. It needs to reference $n/\sqrt{p}$ rows of A and $n/\sqrt{p}$ columns of B.

The task responsible for computing element $c_{i,j}$ of the product matrix requires access to every element of row i of A and every element of column j of B. With a row-oriented agglomeration, every process is responsible for computing elements of entire rows of C, meaning it requires access to every element of B. (See Figure 11.8a.)

If, in contrast, we agglomerate tasks responsible for a square (or nearly square) block of C, the number of elements of B any process needs to access is dramatically reduced.

Let's figure out how much better this scheme is. To simplify the math, let's assume that matrices A, B, and C have dimensions $n \times n$, that p is a square number, and that n is a multiple of $\sqrt{p}$. Each process is responsible for computing an $(n/\sqrt{p}) \times (n/\sqrt{p})$ block of matrix C. To compute these elements, the process needs to reference $n/\sqrt{p}$ rows of A and $n/\sqrt{p}$ columns of B (See Figure 11.8b.

Each process still performs an equal share of the computations—$2n^3/p$. The number of elements each process needs access to is $2n(n/\sqrt{p})$. The computation-to-communication ratio is

$$\frac{2n^3/p}{2n^2/\sqrt{p}} = \frac{n}{\sqrt{p}}$$

Let's determine when the computation-to-communication ratio for Cannon's algorithm is superior to the ratio for the "rowwise" algorithm:

$$\frac{n}{\sqrt{p}} > \frac{2n}{p} \Rightarrow \sqrt{p} > 2 \Rightarrow p > 4$$

Cannon's algorithm seems to hold more promise when the number of processes is greater than four.

11.4.2 Communication

Now that we've established the potential for an algorithm based on a checkerboard block decomposition, let's see if we can unlock that potential. First, let's take a look at how A and B are distributed among the processes in a checkerboard block decomposition (Figure 11.9a). Process $P_{i,j}$ contains blocks $A_{i,j}$ and $B_{i,j}$ and is responsible for computing block $C_{i,j}$. Except for the processes on the main diagonal, processes hold blocks of A and B that do not need to be multiplied.

We need to move the blocks around so that every process $P_{i,j}$ has a pair of blocks whose multiplication will contribute to the calculation of $C_{i,j}$. One way to do this is illustrated in Figure 11.9b. Each process in row i of the process mesh cycles its block of A to the process i places to its left. Each process in column j of the process mesh cycles its block of B to the process j places above it. Now

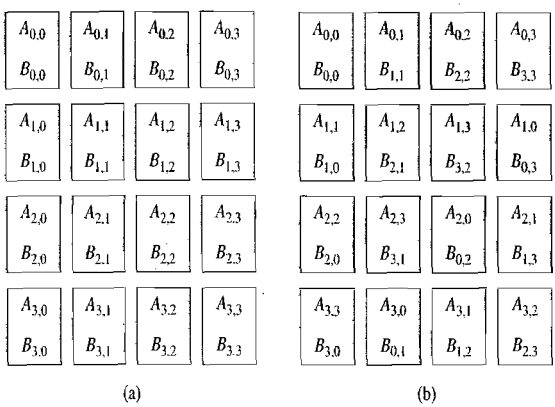


Figure 11.9 Alignment of blocks for matrix multiplication. (a) Initial distribution of blocks among processes. Process $P_{i,j}$ contains blocks $A_{i,j}$ and $B_{i,j}$. The block matrix multiplication algorithm multiplies all pairs $A_{i,k}B_{k,j}$. Note that in the original distribution only the processes on the main diagonal ($P_{0,0}$, $P_{1,1}$, $P_{2,2}$, and $P_{3,3}$) have such pairs. (b) The parallel algorithm cycles each row i of A to the left by i column positions. It cycles each column i of matrix B upward by i row positions. Now every processor $P_{i,j}$ has a pair of blocks to multiply.

we've satisfied our condition: each process can multiply the blocks of A and B it controls to produce a partial result for its block of C.

Recall the size of the process mesh is $\sqrt{p} \times \sqrt{p}$. After the initial step to rearrange the blocks of A and B, the parallel checkerboard matrix multiplication algorithm has $\sqrt{p}$ steps. Each process multiplies the blocks of A and B it controls, adding the result to its partial sum of C. It cycles its block of A to the process to its left, and it receives a new block of A from the process on its right. It cycles its block of B to the process above it, and it receives a new block of B from the process below it. Figure 11.10 illustrates this block-cycling activity from the point of view of process $P_{1,2}$ in a 4×4 process mesh.

11.4.3 Analysis

In this subsection we'll derive an expression for the expected execution time of Cannon's algorithm. To simplify our analysis, we assume that A, B, and C are all $n \times n$ matrices. We also assume that p is a square number and that n is a multiple of $\sqrt{p}$, the number of active processes. Each process is responsible for computing an $(n/\sqrt{p}) \times (n/\sqrt{p})$ portion of C.

First let's consider the computation time. When the algorithm begins, each process initializes its portion of C to 0. During each iteration, every process multiplies an $(n/\sqrt{p}) \times (n/\sqrt{p})$ block of A by an $(n/\sqrt{p}) \times (n/\sqrt{p})$ block of B and adds the result to its partial result for C. If χ is the time needed for one of the add-multiply steps inside an inner product, the computational time of each iteration is

$$\chi(n/\sqrt{p})^3 = \chi n^3/p^{3/2}$$

The algorithm has $\sqrt{p}$ iterations. Hence the total computation time is (as we should expect)

$$\sqrt{p}\chi n^3/p^{3/2} = \chi n^3/p$$

Now let's look at the communication requirements. Before the first iteration, each process must send its blocks of A and B to the appropriate destination processes and receive the blocks of A and B it needs for the first iteration. Our model assumes that messages may be sent and received concurrently, but it allows only a single message at a time to be sent or received. Let $1/\beta$ be the time needed to transmit a single matrix element. The time needed for the initial block distribution is

$$2\left(\lambda + \frac{n^2}{p\beta}\right)$$

During each of the $\sqrt{p}$ iterations, every process must pass along its A and B blocks and receive new blocks to multiply. The total time required for these steps is

$$2\sqrt{p}\left(\lambda + \frac{n^2}{p\beta}\right)$$

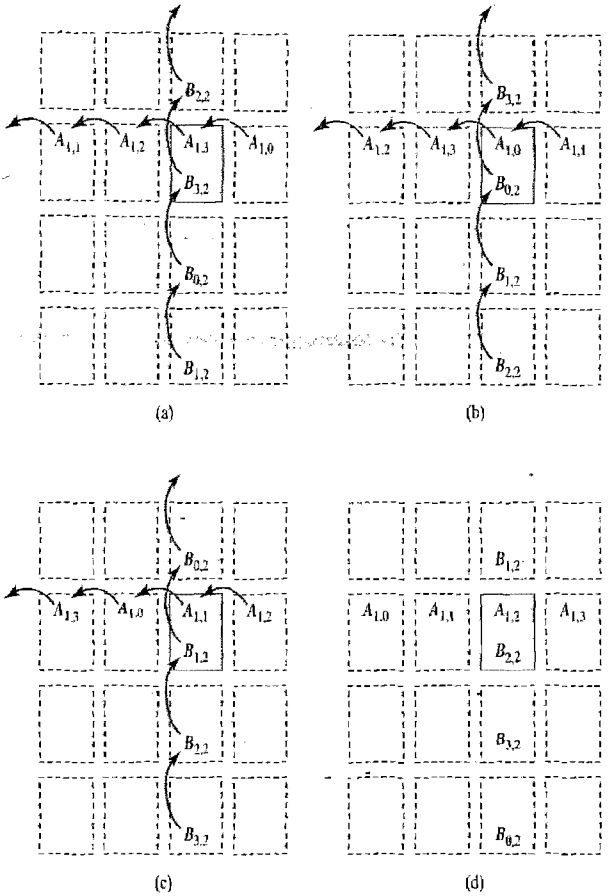


Figure 11.10 Cannon's matrix multiplication algorithm from the point of view of process $P_{1,2}$. Note that processes are organized into a 2-D mesh, and each process has already sent its blocks of A and B to the process that needs them for the first iteration. (a) First block multiplication step. After each block multiplication process $P_{1,2}$ sends its block of A to the process on its left and receives a new block of A from the process on its right. Similarly, it sends its block of B to the process above it and receives a new block of B from the process below it. (b) Second block matrix multiplication step. (c) Third block matrix multiplication step. (d) Final block matrix multiplication step. Summing the results of all block matrix multiplications yields $C_{1,2}$.

Adding these three terms, our expression for the expected overall execution time of Cannon's algorithm is

$$\chi n^3/p + 2(\sqrt{p}+1)\left(\lambda + \frac{n^2}{p\beta}\right)$$

What is the isoefficiency of Cannon's algorithm? The sequential algorithm has time complexity $\Theta(n^3)$. The communication complexity of the parallel algorithm is $\Theta(n^2/\sqrt{p})$. We multiply the community complexity by the number of processors p to get the overhead term: $T_o(n, p) = \Theta(\sqrt{p}n^2)$. Hence the isoefficiency relation for the rowwise block-striped matrix multiplication algorithm is

$$n^3 \geq C\sqrt{p}n^2 \Rightarrow n \geq C\sqrt{p}$$

Recall $M(n) = n^2$. Hence the scalability function is:

$$M(C\sqrt{p})/p = C^2p/p = C^2$$

Because constant memory utilization per processor is sufficient to maintain efficiency as processors are added, we conclude Cannon's algorithm is highly scalable.

As in the case of the row-oriented algorithm, Cannon's algorithm presents a good opportunity for overlapping communications with computation. If there is enough memory to buffer new A and B blocks while working on the current blocks, each process can initiate its sends and receives of A and B blocks before starting the matrix multiplication for that iteration. After the matrix multiplication step, the process can check for the completion of the message receives before starting the next iteration. Since the communication complexity is $\Theta(n^2)$ and the computational complexity is $\Theta(n^3)$, the communication step can be almost completely overlapped with computations when the matrix sizes are large enough.

11.5 SUMMARY

In this chapter we have developed two parallel algorithms for matrix multiplication. The first algorithm is based on a rowwise block-striped matrix decomposition, while the second (Cannon's algorithm) is based on a checkerboard block matrix decomposition. Both algorithms divide the computations evenly among the processes. Cannon's algorithm, however, requires less communication among processes. Isoefficiency analysis reveals that Cannon's algorithm is highly scalable, while the first is not. If sufficient memory is available, both algorithms can benefit from communication/computation overlapping.

We also explored performance issues related to sequential matrix multiplication. The straightforward algorithm has a memory reference pattern that results in a poor cache hit rate once the second factor matrix no longer fits in cache. We presented a recursive matrix multiplication algorithm that divides matrices into blocks when the matrices are too large to fit in cache. We showed how a program

based on this algorithm maintains high CPU performance, even as the matrix sizes grow beyond the cache limits.

In order to achieve best performance, parallel programs performing matrix multiplication should rely upon a high-speed sequential matrix multiplication function, such as the recursive function presented in this chapter, when multiplying submatrices.

11.6 KEY TERMS

Cannon's algorithm

11.7 BIBLIOGRAPHIC NOTES

In this chapter we showed how a recursive matrix multiplication algorithm led to an improved cache hit rate. Recursion is often an effective variable blocking technique for dense linear algebra algorithms, as pointed out by Gustavson [48].

11.8 EXERCISES

11.1 Suppose $A = \begin{pmatrix} 1 & 2 & -3 & -2 \\ 4 & 1 & -1 & 3 \\ 3 & 2 & 1 & -4 \end{pmatrix}$ and $B = \begin{pmatrix} -2 & -3 \\ 3 & 2 \\ 4 & -1 \\ 1 & -4 \end{pmatrix}$

a. Compute $C = AB$.

b. Consider the submatrices

$$A_{00} = \begin{pmatrix} 1 & 2 \end{pmatrix} \quad A_{01} = \begin{pmatrix} -3 & -2 \end{pmatrix}$$

$$A_{10} = \begin{pmatrix} 4 & 1 \\ 3 & 2 \end{pmatrix} \quad A_{11} = \begin{pmatrix} -1 & 3 \\ 1 & -4 \end{pmatrix}$$

and

$$B_{00} = \begin{pmatrix} -2 \\ 3 \end{pmatrix} \quad B_{01} = \begin{pmatrix} -3 \\ 2 \end{pmatrix}$$

$$B_{10} = \begin{pmatrix} 4 \\ 1 \end{pmatrix} \quad B_{11} = \begin{pmatrix} -1 \\ -4 \end{pmatrix}$$

Compute $C = \begin{pmatrix} A_{00}B_{00} + A_{01}B_{10} & A_{00}B_{01} + A_{01}B_{11} \\ A_{10}B_{00} + A_{11}B_{10} & A_{10}B_{11} + A_{11}B_{11} \end{pmatrix}$.

Show the result of each block matrix multiplication.

11.2 In the parallel matrix multiplication algorithm based upon a rowwise block-striped matrix decomposition, each process ends up multiplying its portion of A by the entire matrix B. If we replicated B across all processes, it would greatly simplify the algorithm. What is the fundamental problem with this approach?

11.3 Both the rowwise algorithm and Cannon's algorithm can call the recursive sequential matrix multiplication algorithm as a subroutine when multiplying their portions of A and B.

a. Why is Cannon's algorithm a better match for the recursive sequential matrix multiplication algorithm than the algorithm based on a rowwise striped decomposition?

b. Design a modification to the recursive sequential matrix multiplication algorithm that addresses the problem raised in part (a).

11.4 Consider the optimization of overlapping communication steps with computation steps in the two parallel matrix multiplication algorithms discussed in this chapter. Suppose $p = 16$, $\beta = 1.5 \times 10^6$/sec, $\lambda = 250$ μsec, and $\chi = 10$ nanosec.

a. For what values of n can we expect the communication time per iteration of the rowwise algorithm to be less than the computation time?

b. For what values of n can we expect the communication time per iteration of Cannon's algorithm to be less than the computation time?

11.5 Write a program implementing the parallel matrix multiplication algorithm described in Section 11.3. The program should read the factor matrices from files and write the product matrix to a file; the names of the files should be specified on the command line. Assume the matrices contain double-precision floating-point values. They should be stored in the files according to the protocol first described in Chapter 6: two integers m and n, indicating the number of matrix rows and columns, respectively, followed by mn double-precision floating-point values.

a. Benchmark your program on $1, 2, 3, 4, \ldots, p$ processors for square matrices of size 100, 200, 400, and 800, ignoring file I/O time. Plot the four speedup curves on a graph.

b. Benchmark your program on $1, 2, \ldots, p$ processors for square matrices of size 100, 200, 400, and 800, taking into account file I/O time. Plot the four speedup curves on a graph.

11.6 Write a program implementing Cannon's algorithm described in Section 11.4, assuming that the number of processes executing the program is a square number. The program should read the factor matrices from files and write the product matrix to a file; the names of the files should be specified on the command line. Assume the matrices contain double-precision floating-point values. They should be stored in the files according to the protocol first described in Chapter 6: two integers m and n, indicating the number of matrix rows and columns, respectively, followed by mn double-precision floating-point values.

a. Benchmark your program on $1, 4, 9, \ldots, p$ processors for square matrices of size 100, 200, 400, and 800, ignoring file I/O time. Plot the four speedup curves on a graph.

b. Benchmark your program on $1, 4, 9, \ldots, p$ processors for square matrices of size 100, 200, 400, and 800, taking into account file I/O time. Plot the four speedup curves on a graph.

11.7 Design a version of Cannon's algorithm that works when the number of processes is not a square number.

11.8 Write a parallel program based on Cannon's algorithm that takes advantage of all processes available, even when the number of processes is not a square number. The program should read the factor matrices from files and write the product matrix to a file; the names of the files should be specified on the command line. Assume the matrices contain double-precision floating-point values. They should be stored in the files according to the protocol first described in Chapter 6: two integers m and n, indicating the number of matrix rows and columns, respectively, followed by mn double-precision floating-point values.

a. Benchmark your program on $1, 2, 3, 4, \ldots, p$ processors for square matrices of size 100, 200, 400, and 800, ignoring file I/O time. Plot the four speedup curves on a graph.

b. Benchmark your program on $1, 2, 3, 4, \ldots, p$ processors for square matrices of size 100, 200, 400, and 800, taking into account file I/O time. Plot the four speedup curves on a graph.

CHAPTER

12

Solving Linear Systems

Concern for man himself and his fate must always form the chief interest of all technical endeavors, concern for the great unsolved problems of the organization of labor and the distribution of goods—in order that the creations of our mind shall be a blessing and not a curse to mankind. Never forget this in the midst of your diagrams and equations.

Albert Einstein, Address at the California Institute of Technology, 1931

12.1 INTRODUCTION

Many scientific and engineering problems can take the form of a system of linear equations. Here is a sampling of the domains from which these problems arise:

- structural analysis (civil engineering)
- heat transport (mechanical engineering)
- analysis of power grids (electrical engineering)
- production planning (economics)
- regression analysis (statistics)

Because linear systems derived from realistic problems are often quite large, there is good reason to learn how to solve them efficiently on parallel computers.

In Section 12.2 we define the terminology to be used in the rest of the chapter. In Sections 12.3 and 12.4 we consider direct methods for solving dense systems of linear equations. We begin with an examination of upper triangular systems, which can be solved using the back substitution algorithm. We then consider how to solve dense systems of linear equations. The Gaussian elimination algorithm transforms a dense system into an upper triangular system, which can then be solved using back substitution. In the course of developing a parallel Gaussian elimination

algorithm, we'll introduce a new kind of reduction, called a tournament, and explain how to implement it in MPI.

The discretization of partial differential equations often results in the creation of sparse systems of linear equations. Iterative methods are more appropriate for these systems than Gaussian elimination. They solve a system of linear equations by generating a series of increasingly better approximations to the solution vector. We introduce the Jacobi method and the Gauss-Seidel method in Section 12.5. These methods slowly converge on the solution. In contrast, the conjugate gradient method, presented in Section 12.6, converges on the solution much more rapidly.

For three of these sequential algorithms—back substitution, Gaussian elimination, and the conjugate gradient method—we develop a pair of parallel algorithms based on different data decompositions. In all three cases we see that there are conditions under which each of the decompositions is preferable.

12.2 TERMINOLOGY

A **linear equation** in the n variables $x_0, x_1, \ldots, x_{n-1}$ is an equation that can be expressed as

$$a_0x_0 + a_1x_1 + \cdots + a_{n-1}x_{n-1} = b$$

where $a_0, a_1, \ldots, a_{n-1}$ and b are constants.

A finite set of linear equations in the variables $x_0, x_1, \ldots, x_{n-1}$ is called a **system of linear equations** or a **linear system.** A set of numbers $s_0, s_1, \ldots, s_{n-1}$ is a **solution** to a system of linear equations if and only if making the substitutions $x_0 = s_0, x_1 = s_1, \ldots, x_{n-1} = s_{n-1}$ satisfies every equation in the linear system.

A system of n linear equations in n variables

$$\begin{array}{lllll} a_{0,0}x_0 & +a_{0,1}x_1 & +\cdots & +a_{0,n-1}x_{n-1} & = b_0 \\ a_{1,0}x_0 & +a_{1,1}x_1 & +\cdots & +a_{1,n-1}x_{n-1} & = b_1 \\ \cdots & & & & \\ a_{n-1,0}x_0 & +a_{n-1,1}x_1 & +\cdots & +a_{n-1,n-1}x_{n-1} & = b_{n-1} \end{array}$$

is usually expressed as $Ax = b$, where A is an $n \times n$ matrix containing the $a_{i,j}$s, and x and b are n-element vectors storing x_is and b_is, respectively.

An $n \times n$ matrix A is **symmetrically banded** with semibandwidth w if

$$i - j > w \Rightarrow a_{i,j} = 0 \quad \text{and} \quad j - i > w \Rightarrow a_{i,j} = 0$$

In other words, all of the nonzero elements of A are on the main diagonal, one of the w adjacent diagonals above the main diagonal, or one of the w adjacent diagonals below the main diagonal.

An $n \times n$ matrix A is **upper triangular** if

$$i > j \Rightarrow a_{i,j} = 0$$

An $n \times n$ matrix A is **lower triangular** if

$$i < j \Rightarrow a_{i,j} = 0$$

A matrix is **strictly diagonally dominant** if

$$|a_{ii}| > \sum_{j \neq i} |a_{ij}|, 0 \leq i < n$$

An $n \times n$ matrix A is **symmetric** if $a_{i,j} = a_{j,i}$ for $0 \leq i, j \leq n-1$.

An $n \times n$ matrix A is **positive definite** if for every nonzero vector x and its transpose x^T, the product $x^T Ax > 0$.

12.3 BACK SUBSTITUTION

Back substitution is an algorithm that solves the linear system $Ax = b$, where A is upper triangular. In this section we'll look at the sequential back substitution algorithm and evaluate different ways to execute it on multiple processors.

12.3.1 Sequential Algorithm

Let's start by looking at an example of the back substitution algorithm in action. Suppose we want to solve the system

$$\begin{aligned} 1x_0 + 1x_1 - 1x_2 + 4x_3 &= 8 \\ -2x_1 - 3x_2 + 1x_3 &= 5 \\ 2x_2 - 3x_3 &= 0 \\ 2x_3 &= 4 \end{aligned}$$

We can solve the last equation directly, since it has only a single unknown. After we have determined that $x_3 = 2$, we can simplify the other equations by removing their x_3 terms and adjusting their values of b:

$$\begin{aligned} 1x_0 + 1x_1 - 1x_2 &= 0 \\ -2x_1 - 3x_2 &= 3 \\ 2x_2 &= 6 \\ 2x_3 &= 4 \end{aligned}$$

Now the third equation has only a single unknown, and a simple division yields $x_2 = 3$. Again, we use this information to simplify the two equations above it:

$$\begin{aligned} 1x_0 + 1x_1 &= 3 \\ -2x_1 &= 12 \\ 2x_2 &= 6 \\ 2x_3 &= 4 \end{aligned}$$

```
Back Substitution:

  a[0..n − 1, 0..n − 1] — coefficient matrix
  b[0..n − 1] — constant vector
  x[0..n − 1] — solution vector

for i ← n − 1 down to 1 do
  x[i] ← b[i]/a[i, i]
  for j ← 0 to i − 1 do
    b[j] ← b[j] − x[i] × a[j, i]
    a[j, i] ← 0 {This line is optional}
  endfor
endfor
```

Figure 12.1 The back substitution algorithm solves $Ax = b$ for x where A is an upper triangular matrix.

We have simplified the second equation to contain only a single unknown, and dividing b_1 by $a_{1,1}$ yields $x_1 = -6$. After subtracting $x_1 \times a_{0,1}$ from b_0 we have

$$\begin{aligned} 1x_0 \qquad\qquad\qquad &= 9 \\ -2x_1 \qquad\qquad &= 12 \\ 2x_2 \qquad &= 6 \\ 2x_3 &= 4 \end{aligned}$$

and it is easy to see that $x_0 = 9$.

Pseudocode for the sequential back substitution algorithm appears in Figure 12.1. The time complexity of this algorithm is $\Theta(n^2)$.

Now let's look for ways to execute this algorithm in parallel. We begin by drawing a data dependence diagram that has one vertex for each of the original matrix and vector elements, plus an additional vertex for each time an element of vector b is assigned a new value. As always, an arc from vertex u to vertex v means that the value of u is used to compute the new value of v. The data dependence diagram appears in Figure 12.2. We use heavy lines to illustrate a critical path through the graph. It is evident from the critical path that the elements of x must be computed one at a time. In other words, we cannot execute the outer for loop in parallel.

However, we can execute the inner for loop in parallel. Each new value of b_j depends only on its previous value, the value of x_i, and the value of $a_{j,i}$.

12.3.2 Row-Oriented Parallel Algorithm

Suppose we associate a primitive task with each row of A and the corresponding elements of x and b. During iteration i the task associated with row j must

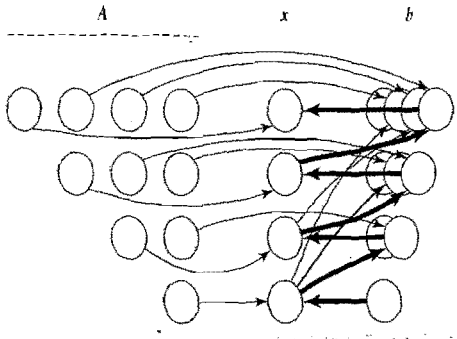


Figure 12.2 Data dependence diagram for the back substitution algorithm. As the algorithm progresses, the values of elements of b get changed. Overlapping circles indicate that the previous value contributes to the new value.

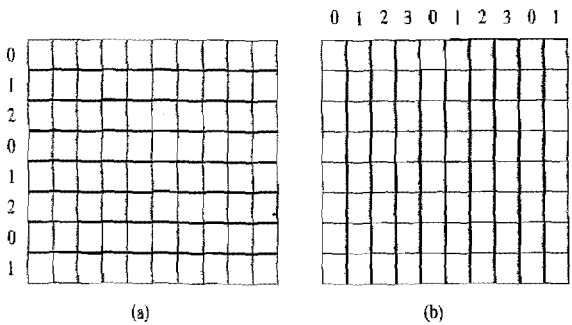


Figure 12.3 Two more ways to decompose a two-dimensional matrix. (The first three ways appeared in Figure 8.3.) (a) Rowwise interleaved striped decomposition. Here eight rows are decomposed among three processes. (b) Columnwise interleaved striped decomposition. Here ten columns are decomposed among four processes.

compute the new value of b_j, meaning it needs access to the current values of x_i and $a_{j,i}$. Since it controls row j of A, it has direct access to $a_{j,i}$. However, it does not have access to x_i unless $i = j$. Hence task i must first compute x_i and then broadcast its value to all of the other tasks.

Let's determine the time complexity of this parallel implementation of back substitution, assuming we agglomerate primitive tasks into p larger tasks (one per process) so that process k controls all rows i where $i \bmod p = k$. We call this a **rowwise interleaved striped decomposition.** It is illustrated in Figure 12.3a.

Over the course of the algorithm the average number of iterations of loop j performed by any process is about $n/(2p)$. Since the algorithm has $n-1$ iterations, the computational complexity of the parallel algorithm is $\Theta(n^2/p)$.

During each iteration the process controlling row i broadcasts x_i to the other processes. Since the algorithm has $n-1$ iterations, the overall message latency is $\Theta(n \log p)$. Because all messages contain a single element, the overall message transmission time is also $\Theta(n \log p)$.

Deriving the isoefficiency relation and scalability function for this algorithm is left as an exercise at the end of the chapter.

12.3.3 Column-Oriented Parallel Algorithm

An alternative design associates one primitive task per column of A. We'll assume that task j, where $0 \leq j < n$, is responsible for column j of A and x_j. At the beginning of the algorithm task $n-1$ is also responsible for vector b.

We agglomerate tasks in an interleaved fashion, creating a columnwise interleaved striped decomposition of the matrix (Figure 12.3b). We can determine the complexity of the parallel algorithm based on this decomposition.

During iteration i process i is responsible for computing x_i and updating vector b. In the first iteration (when $i = n-1$), process $n-1 \bmod p$ already has column $n-1$ of A and vector b, so it may compute x_{n-1} and update b without any communications. However, communications are needed at this point. In the second iteration, process $n-2 \bmod p$ has column $n-2$ of A, but it doesn't have a copy of b (unless $p = 1$). The process that updated b in the first iteration must pass $n-1$ elements of it to the successor process.

For each iteration of the outer loop, one process is responsible for computing x_i and updating b. There is no computational concurrency, and hence the computational complexity of the parallel algorithm is identical to the computational complexity of the sequential algorithm: $\Theta(n^2)$. Between iterations, elements of b must be sent from one process to another. The average number of elements passed is about $n/2$. Since there are $n-1$ iterations, the overall communication latency is $\Theta(n)$ and the overall message transmission time is $\Theta(n^2)$.

12.3.4 Comparison

The row-oriented parallel back substitution algorithm has computational time complexity $\Theta(n^2/p)$ and message transmission time $\Theta(n \log p)$. In contrast, the column-oriented algorithm has computational time complexity $\Theta(n^2)$ and message transmission time $\Theta(n^2)$. For any fixed value of p, the row-oriented algorithm must eventually prove to have better execution time as n increases without bound.

The row-oriented algorithm has overall message latency of $\Theta(n \log p)$, while the column-oriented algorithm has overall message latency of $\Theta(n)$. Hence for any fixed value of n, the column-oriented algorithm must eventually prove to have better execution time as p increases without bound.

We can draw a graph that illustrates the values of n and p for which each algorithm is superior. The result is Figure 12.4. The row-oriented algorithm

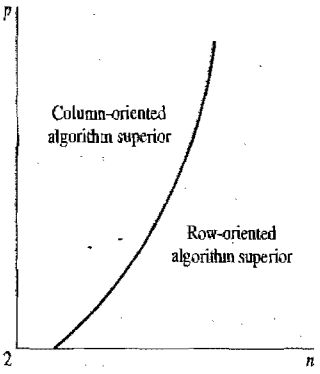


Figure 12.4 A comparison of the row-oriented and column-oriented designs for a parallel back substitution algorithm reveals that each implementation would be superior to the other for certain combinations of n and p.

divides the computational load among the processes, but it requires n broadcast steps. Hence it is superior when n is relatively large and p is relatively small. In contrast, the column-oriented algorithm has no parallelism in the computation, but it requires only n point-to-point messages, making it the preferred algorithm when n is relatively small and p is relatively large.

12.4 GAUSSIAN ELIMINATION

12.4.1 Sequential Algorithm

Gaussian elimination is a well-known algorithm for solving the linear system $Ax = b$ when the matrix A has nonzero elements in arbitrary locations. Gaussian elimination reduces $Ax = b$ to an upper triangular system $Tx = c$, at which point back substitution can be performed to solve for x.

We may perform three operations on a system of linear equations without changing the value of the solution [4]:

- Multiply every term of an equation by a nonzero constant
- Interchange two equations
- Add a multiple of one equation to another equation

Hence we can replace any row of a linear system by the sum of that row and a nonzero multiple of any row of the system.

Let's look at an example. Here is a dense system of linear equations that we want to get into upper triangular form:

$$\begin{aligned} 4x_0 &+6x_1 &+2x_2 &-2x_3 &= 8 \\ 2x_0 & &+5x_2 &-2x_3 &= 4 \\ -4x_0 &-3x_1 &-5x_2 &+4x_3 &= 1 \\ 8x_0 &+18x_1 &-2x_2 &+3x_3 &= 40 \end{aligned}$$

Coefficient $a_{1,0} = 2$ and $a_{0,0} = 4$. Dividing 2 by 4 yields 0.5. If we replace row 1 by the sum of row 1 and -0.5 times row 0, the first term of row 1 becomes 0. Similarly, if we replace row 2 by the sum of row 2 and 1 times row 0, the first term of row 2 becomes 0. Replacing row 3 by the sum of row 3 and -2 times row 0 causes the first term of row 3 to become 0:

$$\begin{aligned} 4x_0 &+6x_1 &+2x_2 &-2x_3 &= 8 \\ &-3x_1 &+4x_2 &-1x_3 &= 0 \\ &+3x_1 &-3x_2 &+2x_3 &= 9 \\ &+6x_1 &-6x_2 &+7x_3 &= 24 \end{aligned}$$

Now that we've driven to 0 all coefficients below $a_{0,0}$, let's focus on coefficient in the column below $a_{1,1}$. We replace row 2 by the sum of row 2 and 1 times row 1. We replace row 3 by the sum of row 3 and 2 times row 1. Here is the resulting system:

$$\begin{aligned} 4x_0 &+6x_1 &+2x_2 &-2x_3 &= 8 \\ &-3x_1 &+4x_2 &-1x_3 &= 0 \\ & & &+1x_3 &= 9 \\ & & &+2x_3 &= 24 \end{aligned}$$

Finally, we need to drive to 0 the coefficient below $a_{2,2}$. We replace row 3 by the sum of row 3 and -2 times row 2:

$$\begin{aligned} 4x_0 &+6x_1 &+2x_2 &-2x_3 &= 8 \\ &-3x_1 &+4x_2 &-1x_3 &= 0 \\ & & &+1x_3 &= 9 \\ & & & &+3x_3 &= 6 \end{aligned}$$

This completes our transformation of the dense linear system into an upper triangular system. At this point we can use back substitution to transform the system into diagonal form, allowing us to determine the solution vector.

Figure 12.5 illustrates one iteration of the algorithm. All nonzero elements below the diagonal and to the left of column i have already been eliminated. In step i the nonzero elements below the diagonal in column i are eliminated by replacing each row j, where $i + 1 \leq j < n$, with the sum of row j and $-a_{j,i}/a_{i,i}$ times row i. After $n - 1$ such iterations, the linear system is upper triangular.

In the straightforward Gaussian elimination algorithm just described, during iteration i row i is the **pivot row**, that is, the row used to drive to zero all nonzero

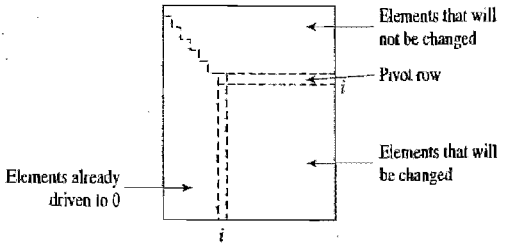


Figure 12.5 Iteration i of the Gaussian elimination algorithm drives to 0 all elements of column i below row i. For each row j below row i, it subtracts a multiple of row i from row j. The multiple is chosen so that after the subtraction the element is column i is 0.

elements below the diagonal in column i. However, if the pivot element $a_{i,i}$ is close to zero, dividing by it can result in significant roundoff errors. Hence this approach does not in general exhibit good numerical stability on digital computers.

Fortunately, a simple variant, called **Gaussian elimination with partial pivoting,** does produce reliable results. In step i of Gaussian elimination with partial pivoting of rows, rows i through $n-1$ are searched for the row whose column i element has the largest absolute value. This row is swapped (pivoted) with row i. Once this has been done, the algorithm uses multiples of the pivot row, now stored as row i, to reduce to zero all nonzero elements of column i in rows $i+1$ through $n-1$. See Figure 12.6.

A sequential algorithm to perform Gaussian elimination with partial pivoting of rows followed by back substitution appears in Figure 12.7. The algorithm has two notable features. First, note that there is no separate array to hold vector b. Since the manipulations of the elements of b are identical to the manipulations of the elements of A, we adjoin b to A, creating an **augmented matrix** with n rows and $n+1$ columns. Hence in this algorithm array a represents the augmented matrix.

Second, note that rather than actually swapping the pivot row and row i in each iteration, the algorithm makes use of indirection. Array element $loc[i]$ contains the index of the pivot row of iteration i.

12.4.2 Parallel Algorithms

Let's determine how well-suited Gaussian elimination is to parallelization. The sequential algorithm requires about $2n^3/3$ floating-point operations [95]. Most of these operations occur inside the innermost for loop. A study of the algorithm's data dependences reveals that both the innermost for loop indexed by k and the middle for loop indexed by j can be executed in parallel. In other words, once the pivot row has been found, the modifications to all unmarked rows may occur simultaneously. Within each row, once the multiplier $a[loc[j], i]/a[loc[i], i]$ has

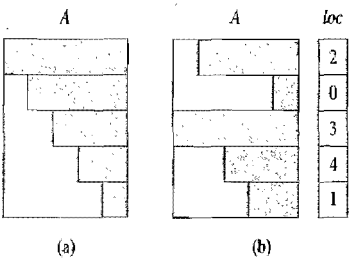


Figure 12.6 Comparing simple Gaussian elimination and Gaussian elimination with partial pivoting. (a) Simple Gaussian elimination transforms coefficient matrix A into an upper triangular matrix. (b) Gaussian elimination with partial pivoting transforms coefficient matrix A into an upper triangular matrix with permuted rows. Array element $loc[i]$ indicates where to find row i of the triangular matrix. For example, array element $loc[0] = 2$ means the 0th row of the triangular matrix is stored in row 2.

been computed, modifications to elements $i + 1$ through $n - 1$ of each row may occur simultaneously. Hence the algorithm is well suited to parallelization.

We will consider two parallel implementations, based on two different data decompositions.

12.4.3 Row-Oriented Algorithm

Let's associate a primitive task with each row of A and the corresponding elements of b and x. If we examine the data dependences for iteration i, we see that determining the pivot row *picked* requires a kind of reduction of the values in column i, which are distributed among the tasks.

We call the interaction to determine the pivot row a **tournament,** because we are interested in the identity of the pivot row (the winner) more than the magnitude of the value stored at column i in the pivot row (the score).

How can we implement a tournament in MPI? One way to do it would be to perform two all-reductions. In the first all-reduction, every task from an unmarked row would contribute the absolute value of the column i element of its row of A. (If a task's row had already been used as a pivot row, it would contribute the value 0, to ensure the row isn't chosen again.) After the first all-reduce step, every task would know the maximum value contributed by any task. Now it's time for the second reduction step. Each task compares its value with this "winning" value. If its value matches the winning value, it contributes its ID number; otherwise, it

```
Gaussian Elimination (Row Pivoting):
  for i ← 0 to n − 1
    loc[i] ← i
  endfor

  for i ← 0 to n − 1

    {Find pivot row picked}
    magnitude ← 0
    for j ← i to n − 1
      if |a[loc[j], i]| > magnitude
        magnitude ← |a[loc[j],i]|
        picked ← j
      endif
    endfor
    tmp ← loc[i]
    loc[i] ← loc[picked]
    loc[picked] ← tmp

    {Drive to 0 column i elements in unmarked rows}
    for j ← i + 1 to n − 1
      t ← a[loc[j], i]/a[loc[i], i]
      for k ← i + 1 to n + 1
        a[loc[j], k] ← a[loc[j], k] − a[loc[i], k] × t
      endfor
    endfor
  endfor

  {Back substitution}
  for i ← n − 1 down to 0
    x[i] ← a[loc[i], n]/a[loc[i], i]
    for j ← 0 to i − 1 do
      a[loc[j], n] ← a[loc[j], n] − x[i] × a[loc[j], i]
    endfor
  endfor
```

Figure 12.7 Sequential Gaussian elimination algorithm with partial pivoting, followed by back substitution.

contributes −1. After another all-reduce step with the maximum operator, every task knows the ID number of a task with the largest value. (We say *a* task rather than *the* task, because more than one task may tie for having the largest value in column i.)

While this works, it seems wasteful to perform two all-reductions, one right after the other. Fortunately, MPI provides a way to implement a tournament in a single all-reduction. The operator `MPI_MAXLOC`, applied to a sequence of p pairs $(v_0, i_0), (v_1, i_1), \ldots, (v_{p-1}, i_{p-1})$, finds the maximum value v_k among $v_0, v_1, \ldots, v_{p-1}$ and returns the pair (v_k, i_k).

In order to use `MPI_MAXLOC` (or its analog `MPI_MINLOC`) in a reduce operation, you must provide a datatype that represents a (value, index) pair. MPI provides six predefined (value, index) datatypes. They are shown in Table 12.1.

Table 12.1 MPI datatypes representing (value, index) pairs. Note that the index must always be a variable of type int.

MPI datatype	Meaning
MPI_2INT	Two ints
MPI_DOUBLE_INT	A double followed by an int
MPI_FLOAT_INT	A float followed by an int
MPI_LONG_INT	A long followed by an int
MPI_LONG_DOUBLE_INT	A long double followed by an int
MPI_SHORT_INT	A short followed by an int

We need to create a C structure to contain the (value, index) pair. We pass the structure to the reduce function. Here is how we could use this feature in a parallel implementation of Gaussian elimination:

```
struct {
   double value;
   int    index;
} local, global;

...

local.value = fabs(a[j][i]);
local.index = j;

...

MPI_Allreduce (&local, &global, 1, MPI_DOUBLE_INT,
               MPI_MAX_LOC, MPI_COMM_WORLD);
```

Every process in the communicator passes its (value, index) pair to `MPI_Allreduce` through struct `local`. When the function returns, the maximum value and the index associated with that value are in struct `global`.

Determining the pivot row during iteration i happens in two steps. First, each process finds, among the unmarked rows it is responsible for, the row having the largest magnitude value in column i. This has time complexity $\Theta(n/p)$. Second, the processes participate in a tournament to find the pivot row. The tournament has time complexity $\Theta(\log p)$.

That is the first communication step needed per iteration, but there is another. See Figure 12.8. In order to compute the new value of $a[j, k]$, a task needs access to the values of $a[j, i]$, $a[picked, i]$, and $a[picked, k]$. We've assigned each task a row of A, so the task controlling $a[j, k]$ also controls $a[j, i]$, but the values of $a[picked, i]$ and $a[picked, k]$ are held by another task. Hence a broadcast step is also needed.

The task controlling row *picked* could broadcast $a[picked, k]$ to the other tasks for each iteration of the for loop indexed by k, but this would result in $O(n)$

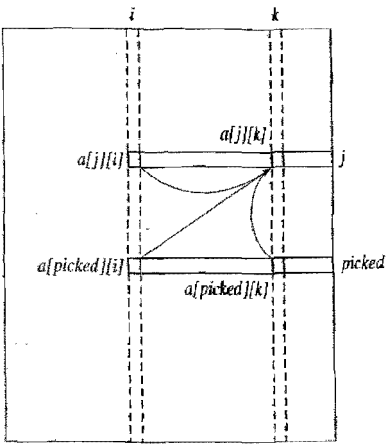


Figure 12.8 In order to update *a[j][k]*, a task needs to have the values of *a[j][i]*, *a[picked][i]*, and *a[picked][k]*.

broadcast steps per iteration. It makes more sense for the broadcast to take place before this for loop. In other words, elements i through n of row *picked* should be broadcast at once to the other tasks. The average number of elements being broadcast is about $n/2$. Hence the message latency of the broadcast is $\Theta(\log p)$, and the message transmission time of the broadcast is $\Theta(n \log p)$.

Combining both communication steps, we see that the row-oriented parallel Gaussian elimination algorithm has overall message latency $\Theta(n \log p)$ and overall message transmission time $\Theta(n^2 \log p)$.

We still need to decide how to agglomerate the primitive tasks into larger tasks that can be associated with MPI processes. Using a rowwise block-striped decomposition is a sound strategy. The use of partial pivoting means that (in the absence of other information) one unmarked row has as great a chance as any other of being chosen as the pivot row for a particular iteration. As the algorithm progresses, the expected number of unmarked rows per process will remain balanced, and the computational complexity of the parallel algorithm is $\Theta(n^3/p)$.

Let's determine the isoefficiency of this parallel Gaussian elimination algorithm. The total communication overhead across p processes is $\Theta(n^2 p \log p)$. Hence

$$n^3 \geq Cn^2 p \log p \Rightarrow n \geq Cp \log p$$

Let's find the scalability function of this parallel system. Since $M(n) = n^2$, we have

$$M(Cp \log p)/p = C^2 p^2 \log^2 p/p = C^2 p \log^2 p$$

This algorithm has poor scalability.

12.4.4 Column-Oriented Algorithm

Let's look at an alternative design for a parallel Gaussian elimination algorithm. We associate one primitive task with each column of A and another primitive task with vector b.

During iteration i of the algorithm, the task controlling column i of A is responsible for finding the candidate element with the largest magnitude. It must only consider rows that have not yet been used as pivot rows. Hence every task needs a copy of array *loc*.

In a single iteration, the column-oriented algorithm spends $\Theta(n)$ time identifying the pivot row.

After the task responsible for column i has identified the pivot row, it must broadcast the identity of the pivot row and the column i elements of the unmarked rows to the other tasks, which need this information in order to do their share of the updates of A and b. This step has message latency $\Theta(\log p)$ and message transmission time $\Theta(n \log p)$. Over the course of the entire algorithm, the overall message latency is $\Theta(n \log p)$ and the overall message transmission time is $\Theta(n^2 \log p)$.

If we agglomerate the primitive tasks in an interleaved fashion, we end up with a columnwise interleaved striped decomposition of A. This decomposition ensures that the workload remains balanced as the algorithm progresses.

Each processor performs nearly an equal share of the computations each iteration. Hence the computational complexity of the parallel algorithm is $\Theta(n^3/p)$.

The isoefficiency of the column-oriented algorithm is the same as the row-oriented algorithm. The algorithm is not highly scalable.

12.4.5 Comparison

Both the row-oriented and the column-oriented parallel Gaussian elimination algorithms evenly divide the computational work inside the doubly nested for loops indexed by j and k. The row-oriented algorithm requires that the process responsible for the pivot row broadcast it to the other processes. The column-oriented algorithm requires that the process responsible for column i broadcast it to the other processes during iteration i. In these two respects, therefore, the expected execution time of both parallel algorithms should be about the same.

The most significant difference between the two algorithms, therefore, is in the identification of the pivot row. The row-oriented algorithm divides the work to find the pivot row among the processes at the cost of an all-reduce step. The column-oriented algorithm performs this step sequentially; no communication is required. Hence the row-oriented algorithm should be superior when n is relatively larger and p is relatively smaller, while the column-oriented algorithm should be superior when p is relatively larger and n is relatively smaller. Note that this is the same conclusion we reached when we discussed the row-oriented and column-oriented back substitution algorithms.

Neither of these algorithms, however, exhibits good scalability. We need to find a way to reduce the communication overhead.

12.4.6 Pipelined, Row-Oriented Algorithm

The row-oriented and column-oriented algorithms we have just considered are synchronous in the sense that they neatly divide the parallel program's execution into communication and computation phases. Consider the row-oriented algorithm. First the processes participate in a tournament to determine the pivot row. Then the process controlling the pivot rows broadcasts it to the other processes. After the broadcast step, all of the processes use the pivot row to reduce the portions of the submatrices they control. Once this has been done, the processes again participate in a tournament to determine the next pivot row.

The disadvantage of the synchronous approach is that processes are not performing computations during the broadcast steps, and the cumulative time complexity of the broadcasts, $\Theta(n^2/p)$, is large enough to ensure the parallel algorithm has poor scalability.

We need to find a way to overlap communication time with computation time. We could do this if we knew in advance the pivot row for iteration i. Recall that we introduced partial pivoting of rows in order to ensure numerical stability, but by doing this we make it impossible to predict the row that will serve as the pivot row for iteration i. What if instead of applying the partial pivoting principle to the rows of the matrix, we applied partial pivoting to its *columns*? In iteration i we will examine row i to find the element with the largest magnitude. We let this serve as the pivot element. We then reduce rows $i + 1$ through $n - 1$ of the coefficient matrix, zeroing out their elements in the column containing the pivot element. Pseudocode for Gaussian elimination based on partial pivoting of columns appears in Figure 12.9.

Let's design a parallel algorithm from the sequential algorithm based on pivoting of columns. We choose a rowwise interleaved striped decomposition of the augmented matrix, and we organize the processes as a logical ring.

When the algorithm begins execution, process 0 searches row 0 to determine the column containing the element with the largest magnitude. As soon as it finishes the search, it sends a message to task 1 containing row 0 and the index of the pivot element. While this message is being transmitted, process 0 can reduce the rest of its share of the augmented matrix.

Process 1 waits until it has received row 0 from process 0. After receiving row 0, it immediately passes it along to process 2. Then it uses row 0 and information about the pivot element to reduce its share of the matrix. At this point it can determine the pivot element for row 1. It does so and then initiates a send of row 1 to process 2. While these messages are being passed, process 1 can use row 1 to reduce its share of the rows of the matrix.

Row 0 is sent from process 0 to process 1, from process 1 to process 2, and so on until it reaches process $p - 1$. Row 1 is sent from process 1 to process 2, from process 2 to process 3, and continues around the logical ring of processes

```
Gaussian Elimination (Column Pivoting):
 for i ← 0 to n
   loc[i] ← i
 endfor

 for i ← 0 to n − 1

   {Find pivot column picked}
   magnitude ← 0
   for j ← i to n − 1
     if |a[i, loc[j]]| > magnitude
       magnitude ← |a[i, loc[j]]|
       picked ← j
     endif
   endfor
   tmp ← loc[i]
   loc[i] ← loc[picked]
   loc[picked] ← tmp

   {Drive to 0 column loc[i] elements in rows i + 1 through n − 1}
   for j ← i + 1 to n − 1
     t ← a[j, loc[i]] / a[i, loc[i]]
     for k ← i to n + 1
       a[j, loc[k]] ← a[j, loc[k]] − a[i, loc[k]] ×t
     endfor
   endfor
 endfor

 {Back substitution}
 for i ← n − 1 down to 0
   x[loc[i]] ← a[i, n] / a[i, loc[i]]
   for j ← 0 to i − 1 do
     a[j, n] ← a[j, n] − x[loc[i]] × a[j, loc[i]]
   endfor
 endfor
```

Figure 12.9 Sequential Gaussian elimination algorithm with pivoting of columns, followed by back substitution.

until it reaches process 0. Each row sent by a process works its way around the ring until it reaches the process's predecessor.

Our previous two parallel implementations of Gaussian elimination rely on broadcasts. This implementation replaces the broadcast step with a series of point-to-point messages being sent around a ring of processes. Why is this approach superior? By pipelining the flow of messages, the parallel algorithm has two decided advantages. First, it facilitates asynchronous execution: processes can reduce their portions of the augmented matrix as soon as the pivot rows are available. Second, it allows processes to effectively overlap communication time with computation time.

If n is sufficiently large, it is reasonable to assume that the time spent transmitting row elements overlaps the time spent reducing matrix elements, because the total reduction time is $\Theta(n^3/p)$, while the total message transmission time

is $\Theta(n^2)$. Message start-up time cannot be overlapped with computation. Since a process must send $n - 1$ messages, the total communication time of this algorithm is $\Theta(n)$.

Let's determine the isoefficiency of this parallel system. The sequential time is $\Theta(n^3)$. Parallel overhead is $\Theta(np)$. Hence

$$n^3 \geq Cnp \Rightarrow n \geq \sqrt{Cp}$$

Since $M(n) = n^2$, our scalability function is:

$$M(\sqrt{Cp})/p = Cp/p = C$$

Assuming n is large enough to ensure that message transmission time essentially overlaps with computation time, this parallel system is perfectly scalable.

12.5 ITERATIVE METHODS

Gaussian elimination followed by back substitution is an example of a **direct method** for solving a system of linear equations. The algorithm works through a prescribed number of steps, and at the end of the algorithm the value of the solution vector is known.

Gaussian elimination works well when the system of linear equations is dense. However, if we apply Gaussian elimination to a **sparse system** of linear equations (one that has relatively few nonzero elements), the coefficient matrix gradually fills in with nonzero elements. Figure 12.10 illustrates this phenomenon with a small, 9×9 system. (The fill becomes more dramatic when the matrix size increases and the distance between the diagonal stripes of nonzero elements grows.) Element fill is undesirable, because it increases storage requirements and the total operation count.

An **iterative method** is an algorithm that comes up with a series of approximations to the value of the solution. Typically, iterative methods require less storage than direct methods. By avoiding operations on zero elements, they can also save a lot of computations. Often, they are amenable to parallelization. In this section we consider two simple iterative methods for solving a system of linear equations.

Assume we want to solve the linear system of equations $Ax = b$, where the diagonal elements of A are nonzero; that is, $a_{ii} \neq 0$ for $0 \leq i < n$. The **Jacobi method** begins with an initial approximation x^0 to the solution vector. It repeatedly computes a new approximation x^{k+1} from the current approximation x^k using the formula

$$x_i^{k+1} = \frac{1}{a_{ii}}\left(b_i - \sum_{j \neq i} a_{ij} x_j^k\right)$$

The Jacobi method in pseudocode appears in Figure 12.11.

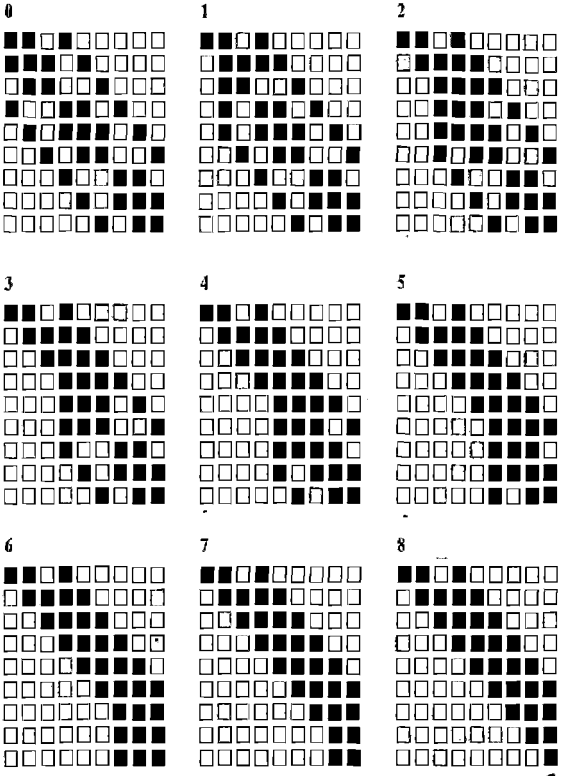


Figure 12.10 Application of Gaussian elimination to a sparse system of nine linear equations. The matrix labeled 0 is the state of the system at the beginning of the algorithm. Black squares represent nonzero coefficients; white squares represent zeroes. The matrices labeled 1 through 8 show the coefficients after each of the 8 iterations of the algorithm.

Figure 12.12 illustrates the successive values of a two-dimensional vector as the Jacobi method solves a system of two linear equations with two unknowns.

Note that in the Jacobi method computing x^{k+1} from x^k is a perfectly parallel operation: each new element of x^{k+1} is computed using the values of x^k. Convergence is quicker if we always use the latest value of x_i that is available. We can accomplish this in our pseudocode algorithm by replacing the line

$$new[j] \leftarrow (1/a[j, j]) \times (b[j] - sum)$$

```
Jacobi Method:

  a[0..n - 1, 0..n - 1] — coefficient matrix
  b[0..n - 1] — constant vector
  new[0..n - 1] — new value of result vector
  sum — accumulates partial results
  x[0..n - 1] — result vector

for i ← 0 to n - 1 do
  x[i] ← 0
endfor
repeat
  for j ← 0 to n - 1 do
    sum ← 0
    for k ← 0 to n - 1 do
      if k ≠ j then
        sum ← sum + a[j,k] × x[k]
      endif
    endfor
    new[j] ← (1/a[j,j]) × (b[j] - sum)
  endfor
  for j ← 0 to n - 1 do
    x[j] ← new[j]
  endfor
until values in x converge
```

Figure 12.11 The Jacobi method is an iterative algorithm for solving $Ax = b$ where the elements of A on the main diagonal are nonzero.

with

$$x[j] \leftarrow (1/a[j, j]) \times (b[j] - sum)$$

and deleting the **for** loop that copies the elements of vector *new* into vector x. The algorithm that results from this change is called the **Gauss-Seidel method.**

Recall that a matrix is strictly diagonally dominant if

$$|a_{ii}| > \sum_{j \neq i} |a_{ij}|, 0 \leq i < n$$

If the coefficient matrix A is strictly diagonally dominant, both the Jacobi method and the Gauss-Seidel method converge on the unique solution to $Ax = b$ for any initial vector x^0.

Even when the Jacobi method and the Gauss-Seidel method are guaranteed to converge on a solution, the rate of convergence is often too slow to make them practical. For this reason, we are not going to develop parallel versions of either algorithm. In Section 12.6 we present an iterative method with much better convergence properties.

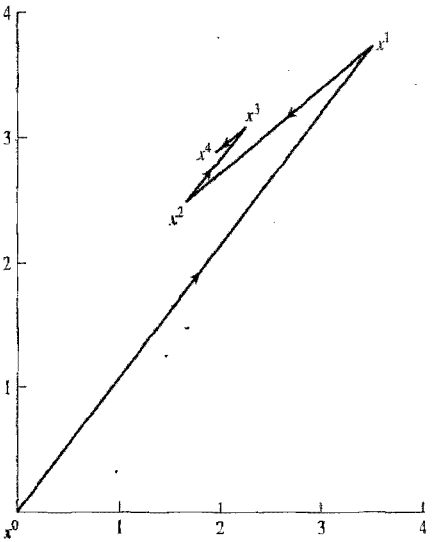


Figure 12.12 Plot of vectors x^1, x^2, x^3, x^4 generated by the Jacobi method as it solves the pair of equations $2x + y = 7$ and $x + 3y = 11$, given $x^0 = (0, 0)$. The successive values of x converge on the solution vector (2, 3).

12.6 THE CONJUGATE GRADIENT METHOD

Recall that $n \times n$ matrix A is positive definite if for every nonzero vector x and its transpose x^T, the product $x^T Ax > 0$. If A is symmetric and positive definite, then the function

$$q(x) = \frac{1}{2}x^T Ax - x^T b + c$$

has a unique minimizer that is the solution to $Ax = b$ [41]. The **conjugate gradient method** is one of many iterative algorithms that solve $Ax = b$ by minimizing $q(x)$. If rounding error is ignored, the conjugate gradient method is guaranteed to converge on a solution in n or fewer iterations [9, 41].

12.6.1 Sequential Algorithm

An iteration of the conjugate gradient method is of the form

$$x(t) = x(t-1) + s(t)d(t)$$

The new value of vector x is a function of the old value of vector x, a scalar step size s, and a direction vector d.

Before iteration 1, values of $x(0)$, $d(0)$, and $g(0)$ must be set. In our implementation of the algorithm $x(0)$ and $d(0)$ are both initialized to the zero vector and $g(0)$ is initialized to $-b$. Every iteration t calculates $x(t)$ in four steps.
Step 1: Compute the gradient

$$g(t) \leftarrow Ax(t-1) - b$$

Step 2: Compute the direction vector

$$d(t) \leftarrow -g(t) + \frac{g(t)^T g(t)}{g(t-1)^T g(t-1)} d(t-1)$$

where $g(t)^T g(t)$ represents the inner product of the transpose of vector $g(t)$ and vector $g(t)$.
Step 3: Compute the step size

$$s(t) \leftarrow -\frac{d(t)^T g(t)}{d(t)^T A d(t)}$$

Step 4: Compute the new approximation of x:

$$x(t) \leftarrow x(t-1) + s(t)d(t)$$

A pseudocode implementation of the conjugate gradient method appears in Figure 12.13.

Figure 12.14 shows how the conjugate gradient method, given the same system of two linear equations as the Jacobi method (Figure 12.12), finds the solution in two iterations.

Suppose matrix A is symmetrically banded with semibandwidth w. (Figure 12.15a). In this case, finding the inner product of a row of A and a vector has time complexity $\Theta(w)$. Hence the matrix-vector multiplication steps have time complexity $\Theta(nw)$. The other vector operations, including the inner product (dot product) operation, have time complexity $\Theta(n)$.

12.6.2 Parallel Implementation

We have discussed parallel algorithms to perform matrix-vector multiplication in Chapter 8. Here we must modify the algorithm to take advantage of the fact that matrix A is banded. In particular, processors only store the portions of the rows of A that contain nonzero elements (Figure 12.15b). This saves memory and makes the algorithm execute faster, but it means that various indices in the matrix-vector multiplication algorithm must be modified.

Suppose we choose a rowwise block-striped decomposition of A and replicate all vectors. In this case the multiplication of A and a vector may be performed without any communications, but an all-gather communication is needed to replicate the result vector. The overall time complexity of the parallel algorithm is $\Theta(n^2 w/p + n \log p)$.

Conjugate Gradient:

```
for i ← 0 to n − 1 do
  d[i] ← 0
  x[i] ← 0
  g[i] ← −b[i]
endfor
for j ← 1 to n do
  d1 ← Inner_Product(g, g)
  g ← Matrix_Vector_Product (A, x)
  for i ← 0 to n − 1 do
    g[i] ← g[i] − b[i]
  endfor
  n1 ← Inner_Product (g, g)
  if n1 < ε break endif
  for i ← 0 to n − 1
    d[i] ← −g[i] + (n1/d1) × d[i]
  endfor
  n2 ← Inner_Product(d, g)
  t ← Matrix_Vector_Product (A, d)
  d2 ← Inner_Product(d, t)
  s ← −n2/d2
  for i ← 0 to n − 1
    x[i] ← x[i] + s × d[i]
  endfor
endfor
```

Figure 12.13 Sequential conjugate gradient algorithm.

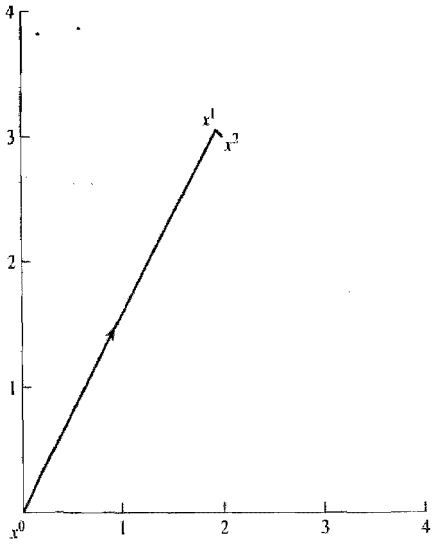


Figure 12.14 The conjugate gradient method requires two iterations to solve the pair of equations $2x + y = 7$ and $x + 3y = 11$. The solution vector is $(2, 3)$.

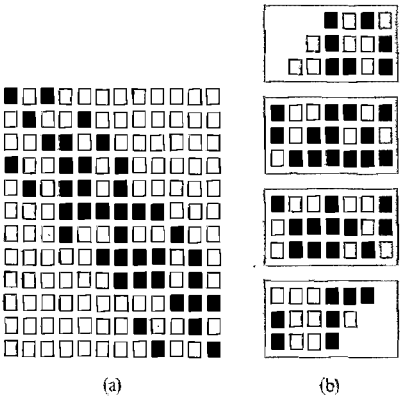


Figure 12.15 Rowwise block-striped decomposition of a symmetrically banded matrix. (a) A symmetrically banded matrix with semibandwidth 3. All nonzero elements (represented by black squares) are on the main diagonal or one of the three diagonals immediately above or below the main diagonal. (b) Storing the matrix on four processors. Since the matrix has semibandwidth 3, each row is represented by $7 = 2 \times 3 + 1$ elements. Note that the fourth (i.e., the middle) entry of each row contains one of the elements on the main diagonal of the matrix.

If, on the other hand, we choose a block decomposition of vectors, an all-gather communication is needed before the matrix-vector multiplication takes place, but no communication is needed to replicate the blocks of the result vector. The overall time complexity of this approach is the same as the first method: $\Theta(n^2 w/p + n \log p)$.

Let's see how the two different data distributions for the vectors affect the complexity of the rest of the algorithm. First let's consider the case where vectors are replicated. Since every process has a complete copy of every vector, it must execute every iteration of every loop updating a vector. Hence the parallel time complexity of the loops modifying values of vectors g, x, and d is $\Theta(n)$. Likewise, the time required to perform the inner product of two n-element vectors is also $\Theta(n)$.

Now let's consider the case where vectors are decomposed by blocks among the processes. In this case the time needed to initialize a vector to 0 or subtract one vector from another is $\Theta(n/p)$. On the other hand, performing an inner product requires that each process find the inner product of its subvector, followed by a sum-reduction step. The complexity of the inner product operation, then, would be $\Theta(n/p + \log p)$.

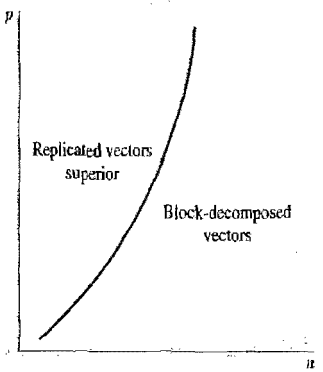


Figure 12.16 Comparison of two approaches to performing inner product (dot product). As $n \to \infty$, distributing the vector elements among processes, performing partial inner products, and then performing a sum-reduction operation is faster than replicating vectors and having each process perform the sequential algorithm. However, for small values of n the time required to perform the reduction is greater than the time saved by distributing the computation, and replication is the preferred alternative.

If we fix p and increase n, eventually the computational time becomes the dominant factor. In this case the algorithm that decomposes the vectors by blocks among the processors is superior. If we fix n and increase p, eventually the communication time becomes the dominant factor. In this case the algorithm that replicates the vectors and avoids all communications is superior. When n dominates p, Figure 12.16 illustrates the regions for which each data distribution scheme is superior.

12.7 SUMMARY

In this chapter we have examined both direct and indirect methods for solving systems of linear equations. We have considered parallel versions of back substitution, Gaussian elimination, and the conjugate gradient method. In each case we have explored two distinct implementations, based on different data decompositions. In each case we have discovered that neither implementation is clearly

superior to the other. Instead, which parallel algorithm is faster depends upon the size of the problem, the number of available processors, the speed of the processors, and the speed of the communication network.

12.8 KEY TERMS

augmented matrix
back substitution
banded matrix
columnwise and rowwise interleaved striped decompositions
conjugate gradient method
direct method
Gaussian elimination
Gauss-Seidel method
iterative method
Jacobi method
linear equation
linear system
lower triangular
partial pivoting
positive definite matrix
pivot row
solution
sparse matrix
strictly diagonally dominant
symmetric
symmetrically banded
system of linear equations
tournament
upper triangular

12.9 BIBLIOGRAPHIC NOTES

The textbook by Bertsekas and Tsitsiklis [9] is the primary source for this chapter. They discuss algorithms to solve systems of linear equations, nonlinear problems, shortest-path problems, and network flow problems, among many others. Consult Golub and Ortega [41] for a thorough, mathematical explanation of how the conjugate gradient method works.

Other books describing parallel numerical algorithms include Dongarra et al. [22] and Fox et al. [33].

Gallivan et al. [36] have surveyed parallel algorithms for dense linear algebra computations.

12.10 EXERCISES

12.1 Use back substitution to solve the upper triangular system produced in Section 12.4.1.

12.2 a. Derive the isoefficiency relation and the scalability function for the row-oriented parallel back substitution algorithm described in Section 12.3.2.

b. Design a parallel back substitution algorithm that uses pipelining to overlap communications with computations. Analyze the time complexity of your algorithm, and determine its isoefficiency relation and scalability function.

12.3 *Forward substitution* is an analog to the back substitution algorithm. It is used to solve lower triangular systems. Write a sequential forward substitution algorithm in pseudocode.

12.4 Implement a C program to solve a system of linear equations $Ax = b$ using Gaussian elimination with row pivoting followed by back substitution. Your program should input the system of equations from a file. The file contains a matrix of doubles in the same format used to store matrices in Chapters 6 and 8. The first two elements of the file are two integers. The first has the value n; the second has the value $n + 1$. The remainder of the file contains $n(n + 1)$ doubles, corresponding to the elements of A and b stored in this order:

$$a_{0,0}, a_{0,1}, \ldots, a_{0,n-1}, b_0,$$
$$a_{1,0}, a_{1,1}, \ldots, a_{1,n-1}, b_1, \ldots,$$
$$a_{n-1,0}, a_{n-1,1}, \ldots, a_{n-1,n-1}, b_{n-1}$$

12.5 Using a C program as your starting point, implement three parallel programs solving a system of linear equations $Ax = b$ using row pivoting. The programs should read the system of equations from a file. The format of the data file is the same as in Exercise 12.4. The programs should print the result vector x to standard output.

a. Use a rowwise block-striped decomposition of the augmented matrix Ab.

b. Use a columnwise interleaved striped decomposition of the augmented matrix Ab. The program should rely on broadcasting to transfer columns.

c. Use a columnwise interleaved striped decomposition of the augmented matrix Ab. The program should use pipelining to overlap communications with computations.

12.6 a. Derive an expected execution time for the row-oriented Gaussian elimination program designed in Section 12.4.

b. Derive an expected execution time for the column-oriented Gaussian elimination program designed in Section 12.4.

c. Using parameters from your parallel computer, draw a graph similar to the one of Figure 12.4 that illustrates the ranges of values of n and p for which programs based on each of the two designs is expected to be superior.

12.7 Implement a C program to solve a system of linear equations $Ax = b$ using Gaussian elimination with column pivoting followed by back substitution. Your program should input the system of equations from a file. The file contains a matrix of doubles in the same format used to store matrices in Chapters 6 and 8. The first two elements of the file are two integers. The first has the value n; the second has the value $n + 1$. The remainder of the file contains $n(n + 1)$ doubles, corresponding to

the elements of A and b stored in this order:

$$a_{0,0}, a_{0,1}, \ldots, a_{0,n-1}, b_0,$$
$$a_{1,0}, a_{1,1}, \ldots, a_{1,n-1}, b_1, \ldots,$$
$$a_{n-1,0}, a_{n-1,1}, \ldots, a_{n-1,n-1}, b_{n-1}$$

12.8 Write a parallel program that implements a pipelined version of Gaussian elimination with column pivoting to solve a system of linear equations $Ax = b$. The augmented matrix Ab should be divided among the processes using a rowwise interleaved striped decomposition. The program should read the system of equations from a file. The format of the data file is the same as in exercise 12.7. The program should at the result vector x to standard output.

12.9 Design a parallel Gaussian elimination algorithm based on a checkerboard block decomposition of the augmented matrix Ab. Determine the isoefficiency of this algorithm and its scalability function.

12.10 Implement a parallel program solving a system of linear equations $Ax = b$, using a checkerboard block decomposition of the augmented matrix Ab. Your program should input the system of equations from a file. The file contains a matrix of doubles in the same format used to store matrices in Chapters 6 and 8. The program should print the solution vector x to standard output.

12.11 Implement a C program to solve a system of linear equations $Ax = b$ using the conjugate gradient method. You can be assured that A is a symmetric, positive definite matrix.

Your program should input the system of equations from a file. The file contains a matrix of doubles in the same format used to store matrices in Chapters 6 and 8. The first two elements of the file are two integers. The first has the value n; the second has the value $n + 1$. The remainder of the file contains $n(n + 1)$ doubles, corresponding to the elements of A and b stored in this order

$$a_{0,0}, a_{0,1}, \ldots, a_{0,n-1}, b_0,$$
$$a_{1,0}, a_{1,1}, \ldots, a_{1,n-1}, b_1, \ldots,$$
$$a_{n-1,0}, a_{n-1,1}, \ldots, a_{n-1,n-1}, b_{n-1}$$

Your program should print the result vector x to standard output.

12.12 The file format we first used in Chapter 6 assumes the matrix is dense. The purpose of this exercise is to develop and exploit a new file format designed for symmetrically banded matrices.

a. Design a file format to store symmetrically banded matrices. The size of a file should be proportional to the number of rows in the matrix times its semibandwidth.

b. Implement a C program to solve a sparse system of linear equations $Ax = b$ using the conjugate gradient method. You can be assured that A is a symmetric, positive definite matrix. Your program should

input matrix A from a file, using the file format you have designed. It should input vector b from another file. Your program should print the result vector to standard output.

c. Using the C program as a starting point, implement a parallel program for the conjugate gradient method that assumes vectors are replicated among processors. Benchmark your program on your parallel computer for various numbers of processors and different problem sizes.

d. Using the C program as a starting point, implement a parallel program for the conjugate gradient method that assumes a block decomposition of vectors among processors. Benchmark your program on your parallel computer for various numbers of processors and different problem sizes.

CHAPTER

13

Finite Difference Methods

Big words do not smite like war-clubs,
Boastful breath is not a bow-string,
Taunts are not so sharp as arrows,
Deeds are better things than words are,
Actions mightier than boastings.

Henry Wadsworth Longfellow, *The Song of Hiawatha*

13.1 INTRODUCTION

An **ordinary differential equation** is an equation containing derivatives of a function of one variable. A **partial differential equation** (PDE) is an equation containing derivatives of a function of two or more variables. Many phenomena studied by scientists and engineers can be modeled by PDEs. Here are a few examples:

- Airflow over an aircraft wing
- Blood circulation in the human body
- Water circulation in an ocean (see Figure 13.1)
- Deformations of a bridge as it carries traffic
- Evolution of a thunderstorm
- Oscillations of a skyscraper as it is hit by an earthquake
- Strength of a toy
- Temperature distribution of a CPU's heat sink
- Vibrations of a subwoofer

It is possible to derive analytical solutions to simple PDEs in simple geometric regions. In general, however, analytical solutions are not possible, and we must seek an approximate result to the equation through numerical (computational)

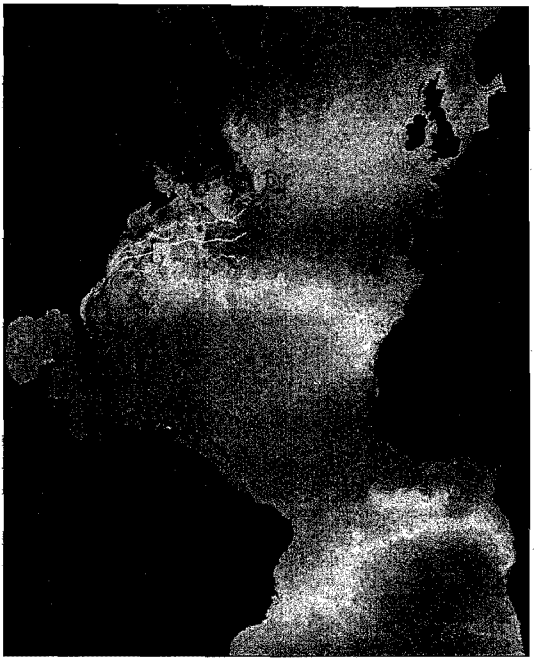


Figure 13.1 Sea surface temperature from a high-resolution (1/12°, approximately 6 kilometer grid spacing on the average) North Atlantic finite difference numerical calculation with the Miami Isopycnic Coordinate Ocean Model (MICOM). (Courtesy MICOM group at the Rosenstiel School of Marine and Atmospheric Science, University of Miami.)

methods. These numerical methods often consume a large number of CPU cycles. That's why it's worthwhile to explore parallel methods for solving PDEs.

The two most common ways to solve PDEs numerically are the finite element method and the finite difference method. This chapter focuses on the finite difference method.

The finite difference method converts a PDE into a matrix equation. As we observed in the previous chapter, the matrices produced by the finite difference method are sparse. (Typically there are only a few nonzero elements per row.) Implementations of the finite difference method fall into one of two broad categories, depending upon how they represent the sparse matrix. Matrix-based implementations represent the matrix explicitly, using data structures that support efficient access of the nonzero elements. In the last chapter we demonstrated how iterative methods can be used to solve these linear equations. Matrix-free implementations represent the matrix values implicitly. In this chapter we focus on matrix-free implementations of the finite difference method.

We begin by defining linear second-order PDEs. Linear second-order PDEs can be put into three categories, and each has different solution methods. We also show how difference quotients approximate the first and second derivatives of a continuous function at a point.

Two case studies—the vibrating string and the steady-state heat distribution problem—illustrate techniques for parallelizing programs.

13.2 PARTIAL DIFFERENTIAL EQUATIONS

13.2.1 Categorizing PDEs

As we have already noted, a PDE is an equation containing derivatives of a function of two or more variables. For example, assume u is a function of x and y: $u = f(x, y)$. We denote the partial derivative of u with respect to x as u_x; similarly, we denote the partial derivative of u with respect to y as u_y. Because only a single partial derivative is taken, these partial derivatives have order one. If k partial derivatives are taken, we say they have order k. Here are the three partial derivatives of u with order two: u_{xx}, u_{xy}, and u_{yy}.

A **second-order partial differential equation** contains no partial derivatives of order more than two. Second-order PDEs are the PDEs most frequently used to solve problems in the physical sciences and engineering.

Linear second-order partial differential equations are of the form

$$Au_{xx} + 2Bu_{xy} + Cu_{yy} + Eu_x + Fu_y + Gu = H \tag{13.1}$$

where A, B, C, D, E, F, G, and H are functions of x and y only.

Here are examples of linear second-order PDEs:

$$4u_x x + 6xyu_{xy} = 0$$

$$\pi u_{xy} + x^2 u_{yy} = \sin(xy)$$

Here are examples of equations that are not linear second-order PDEs:

$$u_{xx}^2 + u_{yy} = 0$$

$$uu_{xy} + \sin(xy)u_{yy} = x + y$$

The first is not a linear second-order PDE because the u_{xx} term is squared; the second is not a linear second-order PDE because the u_{xy} term is multiplied by u.

Based on the values of A, B, and C in equation (13.1), we can classify linear second-order PDEs into three categories:

- **Elliptic PDEs** are those for which $B^2 - AC < 0$.
- **Parabolic PDEs** are those for which $B^2 - AC = 0$.
- **Hyperbolic PDEs** are those for which $B^2 - AC > 0$.

Each of these categories has a well-known representative equation.

The **Poisson equation,** $u_{xx} + u_{yy} = f(x, y)$, is an example of an elliptic PDE. It arises from the study of potential problems in electricity, magnetism, and gravitating matter, steady-state distribution of heat or electricity in homogeneous conductors, and certain fluid flow and torsion problems. When $f(x, y) = 0$, the Poisson equation is called the **Laplace equation.**

The **heat equation,** $ku_{xx} = u_t$, is an example of a parabolic PDE. The heat equation arises from the study of heat conduction in solids. The study of diffusion of liquids and gases results in the same equation as the heat equation, but in this context it is called the **diffusion equation.**

The **wave equation,** $c^2u_{xx} = u_{tt}$, is an example of a hyperbolic PDE. The wave equation arises from modeling wave propagation and the vibration of strings and membranes.

13.2.2 Difference Quotients

While different algorithms are used to solve elliptic, parabolic, and hyperbolic PDEs, all **finite difference methods** approximate the solution to a PDE by dividing the variables (often time and space) into discrete intervals. To illustrate this process, let's consider how to approximate the first and second derivatives of a function.

Consider function f in Figure 13.2. (We're assuming f is a continuous function that has a derivative at each point.) We want to compute the first and second derivatives of f at a particular point x. A reasonable approximation to $f'(x)$ is:

$$f'(x) \approx \frac{f(x + h/2) - f(x - h/2)}{h}$$

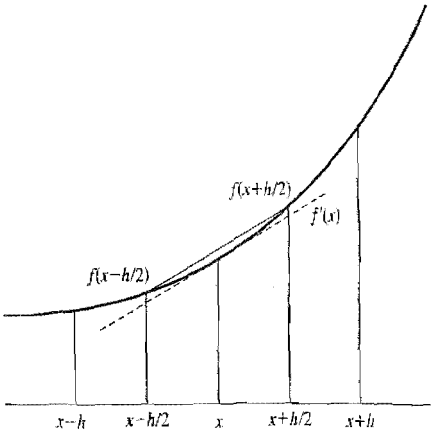


Figure 13.2 Approximating the derivative of function f at x.

By reducing h we can reduce the error in the approximation. We can use this same formula to estimate $f''(x) = f'(f'(x))$:

$$f''(x) \approx \frac{\frac{f(x+h/2-h/2)-f(x+h/2-h/2)}{h} - \frac{f(x-h/2+h/2)-f(x-h/2-h/2)}{h}}{h}$$

$$\approx \frac{f(x+h)-f(x)-((f(x)-f(x-h))}{h^2}$$

$$\approx \frac{f(x+h)-2f(x)+f(x-h)}{h^2}$$

13.3 VIBRATING STRING

As our first case study in the finite difference method, we consider an example of a hyperbolic partial differential equation. This section presents only the briefest sketch of the algorithm's development; see Plybon [92] for more details.

13.3.1 Deriving Equations

Examine Figure 13.3. Our goal is to model the behavior of a vibrating string (such as a guitar string). In particular, we want to be able to determine the position of the string at some future time, based upon its initial position.

The endpoints of the string are fixed. We let variable x represent points along the imaginary line between one endpoint and the other. The left endpoint is where $x = 0$; the right endpoint is where $x = 1$. Hence $0 \leq x \leq 1$.

We let variable t represent time. The initial position of the string is its position at time 0. Hence $t \geq 0$.

Function $u(x, t)$ describes the displacement of the string at point x at time t.

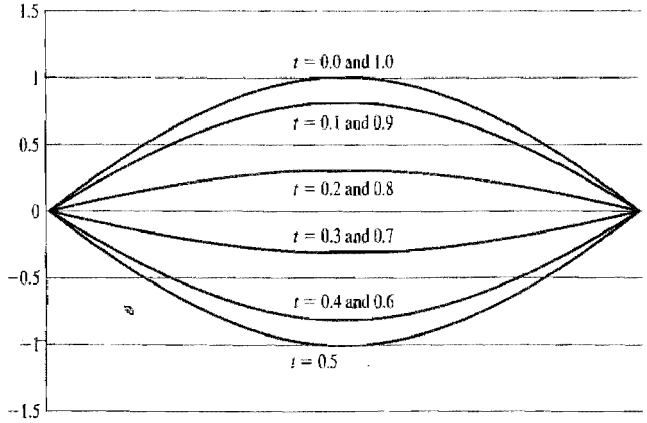


Figure 13.3 Motion of a vibrating string over time.

This particular problem is modeled by a set of equations. The first equation, a linear second-order PDE, expresses how the displacement changes with respect to time:

$$4u_{xx} = u_{tt} \qquad 0 < x < 1, \quad 0 < t$$

The second equation expresses the fact that the string is fixed at both ends:

$$u(0, t) = u(1, t) = 0 \quad \text{where } t > 0$$

The third and fourth equations describe the initial position and velocity of the string at time 0, respectively:

$$u(x, 0) = \sin(\pi x), \quad u_t(x, 0) = 0 \quad \text{where } 0 \le x \le 1$$

Our problem is a specific example of the wave equation, which has this general form:

$$c^2 u_{xx} = u_{tt} \qquad 0 \le x \le a, t \ge 0$$

$$u(x, 0) = F(x) \quad \text{and} \quad u_t(x, 0) = G(x) \quad \text{on} \quad [0, a]$$

$$u(0, t) = u(a, t) = 0$$

In general, we want to find a solution to the problem for values of x between 0 and a, for all times from 0 to T. We divide space into n intervals and time into m intervals, and we define $h = a/n$ and $k = T/m$. In other words, k is the time step and h is the "space step." If the time step k is too large, our discretization will be too crude, and the algorithm will not be stable (i.e., the difference between our approximate solution and the actual solution will grow rapidly with every time step). This is the case when the fraction $kc/h > 1$. On the other hand, if the time step is too small, round-off errors can accumulate, which also reduces the accuracy of the estimate. This is the case when the fraction $kc/h < 1$. Hence the best accuracy is obtained when $kc/h = 1$.

After we have checked to ensure $kc/h \le 1$, we can push ahead. We define

$$x_i = ih \qquad i = 0, 1, \ldots, n$$

$$t_j = jk \qquad j = 0, 1, \ldots, m$$

Now we can define $u_{i,j} = u(x_i, t_j)$ to be the displacement of the string at position x_i and time t_j (see Figure 13.4).

13.3.2 Deriving the Sequential Program

Using the formula we derived in Section 14.2, we can come up with an approximation to the second partial derivative u_{xx}:

$$u_{xx}(x_i, t_j) \approx \frac{u(x_i + h, t_j) - 2u(x_i, t_j) + u(x_i - h, t_j)}{h^2}$$

$$\approx \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{h^2}$$

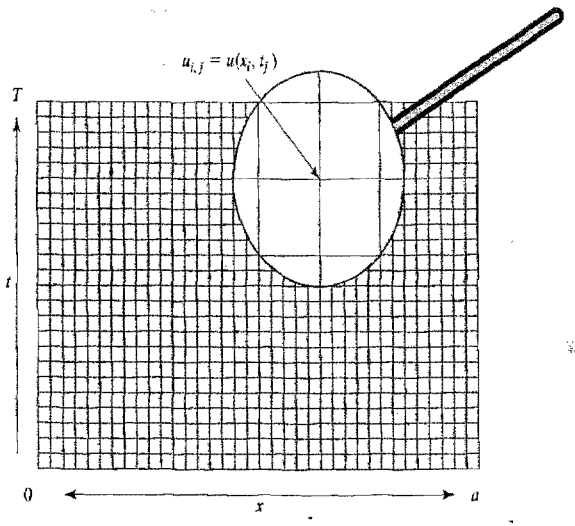


Figure 13.4 Dividing space and time into discrete intervals creates a rectangular lattice. The boundary conditions are along the bottom and sides of the rectangle. Each intersection $u_{i,j}$ represents an approximation of u for x_i and t_j—in other words, the displacement of one point of the string at some point in time.

Similarly, we can approximate the second partial derivative u_{tt}:

$$u_{tt}(x_i, t_j) \approx \frac{u(x_i, t_j + k) - 2u(x_i, t_j) + u(x_i, t_j - k)}{k^2}$$

$$\approx \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{k^2}$$

These approximations can be substituted back into the wave equation. After a series of further approximations and refinements (see Plybon [92] for more details), the C program appearing in Figure 13.5 results. Note: In the process of translating the lattice of Figure 13.4 into the matrix u of Figure 13.5, the subscripts are reversed.

13.3.3 Parallel Program Design

As usual, we begin by choosing primitive tasks, identifying data communication patterns among them, and looking for ways to agglomerate tasks. The finest-grained tasks are those that compute the value of an element of matrix `u`. Examining the C code in Figure 13.5, we find that the value of `u[j+1][i]` depends upon the values of `u[j][i-1]`, `u[j][i]`, `u[j][i+1]`, and `u[j-1][i]`. For a particular value of `u[j+1][i]`, the graph appears in Figure 13.6. If we drew the complete task/channel graph, it would look similar (though not identical) to Figure 3.11a.

```
/* Sequential solution to string vibration problem */

#include <stdio.h>
#include <math.h>
#define F(x) sin(3.14159*(x))   /* Initial string position */
#define G(x) 0.0                /* Initial string velocity */
#define a    1.0                /* Length of string */
#define c    2.0                /* String-related constant */
#define m    20                 /* Discrete time intervals */
#define n    8                  /* Discrete space intervals */
#define T    1.0                /* End time of simulation */

int main (int argc, char *argv[])
{
   double   h;                  /* Space interval length */
   int      i, j;
   double   k;                  /* Time interval length */
   double   L;                  /* Computed coefficient */
   double   u[m+1][n+1];        /* String displacements */

   h = a / n;
   k = T / m;
   L = (k*c/h)*(k*c/h);
   for (j = 0; j <= m; j++) u[j][0] = u[j][n] = 0;
   for (i = 1; i < n; i++) u[0][i] = F(i*h);
   for (i = 1; i < n; i++)
      u[1][i] = (L/2.0)*(u[0][i+1] + u[0][i-1]) +
                (1.0 - L) * u[0][i] + k * G(i*h);
   for (j = 1; j < m; j++)
      for (i = 1; i < n; i++)
         u[j+1][i] = 2.0*(1.0-L)*u[j][i] +
            L*(u[j][i+1] + u[j][i-1]) - u[j-1][i];
   for (j = 0; j <= m; j++) {
      for (i = 0; i <= n; i++) printf ("%6.3f ", u[j][i]);
      putchar ('\n');
   }
   return 0;
}
```

Figure 13.5 C program implementing a finite difference method to solve the string vibration problem (adapted from pseudocode of Plybon [92]).

Computing the displacement at a particular point for different times is inherently sequential: the value of `u[j+1][i]` depends upon `u[j][i]` and `u[j-1][i]`. For this reason we should agglomerate all tasks associated with each value of x_i. At this point all the communications are between adjacent tasks. If we agglomerate tasks associated with contiguous regions of the string, communication among tasks will be minimized.

Suppose the $n+1$ elements of the string are divided among p processes. Let's consider the communications that need to happen when computing row `j+1` of the matrix. See Figure 13.7a. Process q is responsible for computing `u[j+1][i]` for four different values of `i`. It can compute the values of the gray cells without any communications. However, it cannot compute the values of the black cells until it

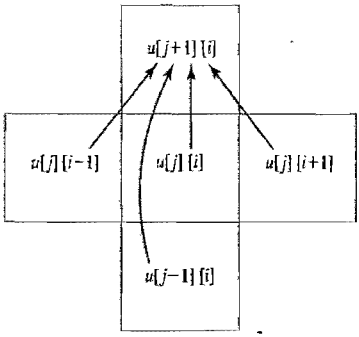


Figure 13.6 The value of `u[j+1][i]` depends upon the values of `u[j][i-1]`, `u[j][i]`, `u[j][i+1]`, and `u[j-1][i]`.

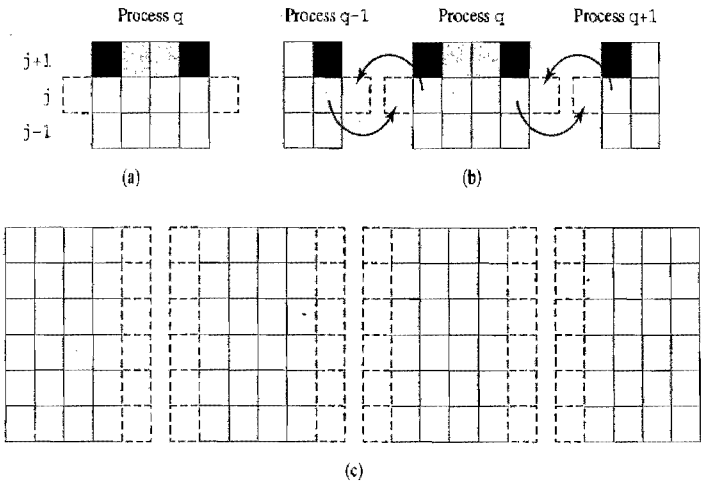


Figure 13.7 Ghost points simplify parallel finite difference programs. (a) When computing row $j+1$, process q has the data values it needs to fill in the gray cells, but it needs values from neighboring processes to fill in the black cells. (b) Every process sends its edge values to its neighbors. Every process receives incoming values into ghost points. Ghost points, then, contain copies of data values. (c) Programming is easier when the ghost points are treated as extra columns or rows in the data array. After the values are received, a single `for` loop can update every element in row `j+1`.

gets values from neighboring processes. In Figure 13.7b we show how process q must exchange values with processes $q-1$ and $q+1$. After these values are received, the black cells may be assigned values.

The parallel program is much easier to code if the same loop that updates the values of the cells on the edge also updates the values of the cells in the interior.

This can be done by allocating two extra columns for process q. These columns will receive the values received from the neighboring processes (one column element per iteration). **Ghost points** are memory locations used to store redundant copies of data held by neighboring processes. Figure 13.7c illustrates how to set up the ghost points when $n = 16$ and $p = 4$, assuming no communications wrap around from one side to the other.

During the iteration that computes row `j+1`, each process sends each of its neighbors the appropriate border value from row `j` and receives the neighbor's row `j` border value in return. After the values have been received into the ghost points, a single `for` loop allows the process to compute all of its row `j+1` values.

13.3.4 Isoefficiency Analysis

The computation time per element is constant, so the sequential time complexity of the algorithm is $\Theta(n)$ per iteration. If each of the p processes has an equal share of the elements, the complexity of the parallel algorithm is $\Theta(n/p)$ per iteration.

During each iteration a typical process must send messages to its two neighbors, each message of length 1, and receive messages from these neighboring processes. The communication time required for these sends and receives is $\Theta(1)$. Hence the overall communication overhead of the parallel algorithm is $\Theta(p)$.

The isoefficiency relation for this algorithm is

$$n \geq Cp$$

While we have described a solution that uses about nm elements to store the position of the string at every point in time, it is possible to implement a solution that uses only about $3n$ memory locations to store values of u as the simulation progresses. Hence $M(n) = n$. The scalability function is

$$M(C_p)/p = C_p/p = C$$

The algorithm we have described is perfectly scalable.

13.3.5 Replicating Computations

What can we do to reduce communication overhead? Since processes are sending messages containing only a single data value, communication time is dominated by the message latency component. We can send two values in virtually the same amount of time it takes to send one value. Let's explore the implications of sending the multiple data values.

Take a look at Figure 13.8. Part (a) illustrates what happens in the algorithm as originally designed. The white squares represent the cells an interior process is responsible for. The dashed white squares are the ghost points. When values are received into the ghost points, the process can compute the values in the gray squares—the values of its section of cells at the next time step.

Part (b) illustrates what we can do if we increase the number of ghost points to two cells on each side. Each process now sends two values to each of its neighboring processes. When it has received a pair of values from each of its

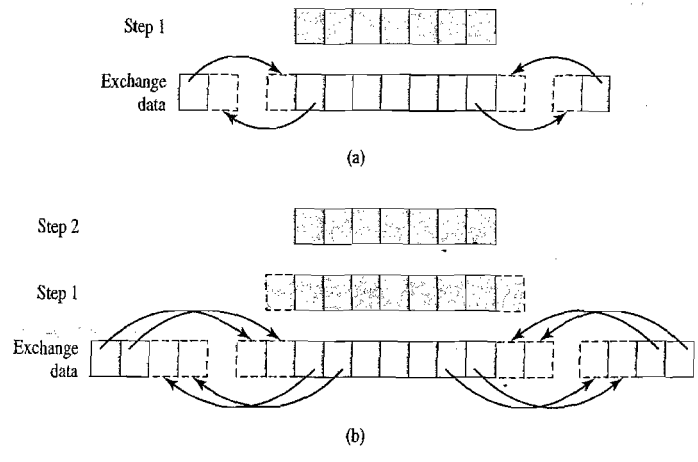


Figure 13.8 Communication time can be reduced by replicating computations. (a) Passing a single edge element allows the algorithm to proceed only a single time step for each communication. (b) If two edge elements are passed, the algorithm can proceed two time steps for each communication, at the cost of two extra computations.

neighbors, it can advance the simulation two time steps. In the first time step it generates values for all the cells it is responsible for, plus values for the ghost points on either side (the gray boxes edged by dashed lines). These are redundant computations, because the value of each of these ghost points is being computed by a neighboring process that is officially responsible for that location. With the redundant values in hand, the process can compute values for the second time step without further message passing.

Increasing the number of ghost points has three effects: increasing message length, reducing message frequency, and adding redundant computations. We want to determine parallel overhead per iteration as a function of the number of ghost points. If there are k ghost points on each side, then messages of length k are sent and received to and from neighboring processes every k iterations. The number of redundant computations added is

$$\sum_{i=1}^{k-1} i = k(k-1)/2$$

The parallel overhead per process per iteration, then, is

$$\frac{2(\lambda + k/\beta) + \chi k(k-1)/2}{k} = \frac{2\lambda}{k} + \frac{2}{\beta} + \frac{\chi(k-1)}{2}$$

The message latency term is inversely proportional to k, while computation time increases quadratically with k. Hence the function typically has the shape

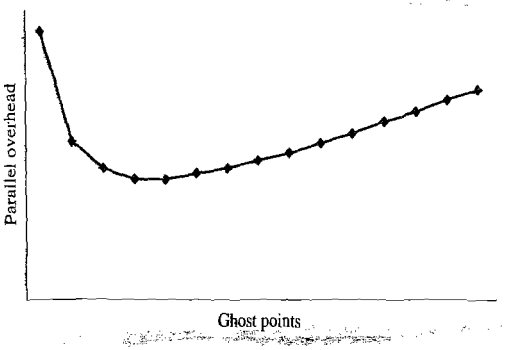


Figure 13.9 Increasing the number of ghost points can reduce parallel overhead up to a point, after which the expense of the redundant computations outweighs the benefits of reducing the number of sends and receives.

illustrated in Figure 13.9. The value of k that minimizes the function depends upon the values of λ and χ.

13.4 STEADY-STATE HEAT DISTRIBUTION

As our second case study, we consider the parallelization of a program that finds the steady-state heat distribution over a thin square plate. (The presentation in this section follows the notation of Plybon [92].)

13.4.1 Deriving Equations

The underlying PDE is the Poisson equation

$$u_{xx} + u_{yy} = f(x, y), \quad 0 \le x \le a, \quad 0 \le y \le b$$

We complete the boundary value problem by adding boundary conditions

$$u(x, 0) = G_1(x) \quad \text{and} \quad u(x, b) = G_2(x) \quad 0 \le x \le a$$
$$u(0, y) = G_3(y) \quad \text{and} \quad u(a, y) = G_4(y) \quad 0 \le y \le b$$

Since the region is rectangular, this is called the **Dirichlet problem.** If functions G_1, G_2, G_3, and G_4 are continuous on the boundaries and function f is continuous inside the rectangle, then the problem has a unique solution.

As in the previous case study, we will create a two-dimensional grid. However, the interpretation of the grid is different in this case study. In the vibrating string problem, each grid point (x_i, t_j) represented the displacement of the string at point x_i at time t_j. In this case study, each grid point (x_i, y_j) represents the value of the steady-state solution at a particular (x, y) location in the rectangle. In the last case study, we only computed the value at each grid point once. In this case

study, we will repeatedly update the values of every interior grid point until the values converge.

13.4.2 Deriving the Sequential Program

We divide spatial dimension x into n pieces and spatial dimension y into m pieces. We define $h = x/n$ and $k = y/m$.

Using the approximation to the second derivative developed in Section 14.2, we determine that

$$u_{xx}(x_i, y_j) \approx \frac{u(x_i + h, y_j) - 2u(x_i, y_j) + u(x_i - h, y_j)}{h^2}$$

$$\approx \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{h^2}$$

Similarly,

$$u_{yy}(x_i, y_j) \approx \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{k^2}$$

If we insert these approximations into the Poisson equation, we get

$$\frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{h^2} + \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{k^2} = f(x_i, y_j)$$

Assume $\lambda = k/h$. After a series of further approximations (detailed in Plybon [92]), we come up with a new formula for the value of the solution at each grid point

$$w_{i,j} = \frac{\lambda^2(w_{i+1,j} + w_{i-1,j}) + w_{i,j+1} + w_{i,j-1} - k^2 f_{i,j}}{2(1 + \lambda^2)}$$

Now let's look at the particular problem we want to solve. A thin steel plate is surrounded on three sides by a condensing steam bath (temperature 100 degrees Celsius). The fourth side touches an ice bath (temperature 0 degrees Celsius). An insulating blanket covers the top and the bottom of the plate. Our goal is to find the steady-state temperature distribution at evenly spaced points within the plate.

Since the points are evenly spaced, $h = k$ and $\lambda = 1$. Since the plate is insulated, no heat is being introduced inside the plate—only on the edges. That means $f_{i,j} = 0$. Hence our finite difference approximation to the solution of the linear second-order PDE reduces to

$$w_{i,j} = \frac{w_{i+1,j} + w_{i-1,j} + w_{i,j+1} + w_{i,j-1}}{4}$$

Starting with initial estimates for all the $w_{i,j}$ values, we can iteratively compute new estimates from previous estimates until the values converge. Relying on the estimates from iteration i to calculate new estimates for iteration $i + 1$ is called the **Jacobi method** [56]. (Note that this is the same algorithm we first encountered in Chapter 12.) A C program implementing a solution to the steady-state heat problem appears in Figure 13.10.

```
/* Sequential Solution to Steady-State Heat Problem */

#include <math.h>
#define N       100
#define EPSILON 0.01

int main (int argc, char *argv[])
{
   double diff;        /* Change in value */
   int    i, j;
   double mean;        /* Average boundary value */
   double u[N][N];     /* Old values */
   double w[N][N];     /* New values */

   /* Set boundary values and compute mean boundary value */
   mean = 0.0;
   for (i = 0; i < N; i++) {
      u[i][0] = u[i][N-1] = u[0][i] = 100.0;
      u[N-1][i] = 0.0;
      mean += u[i][0] + u[i][N-1] + u[0][i] + u[N-1][i];
   }
   mean /= (4.0 * N);

   /* Initialize interior values */
   for (i = 1; i < N-1; i++)
      for (j = 1; j < N-1; j++) u[i][j] = mean;

   /* Compute steady-state solution */
   for (;;) {
      diff = 0.0;
      for (i = 1; i < N-1; i++)
         for (j = 1; j < N-1; j++) {
            w[i][j] = (u[i-1][j] + u[i+1][j] +
                       u[i][j-1] + u[i][j+1])/4.0;
            if (fabs(w[i][j] - u[i][j]) > diff)
               diff = fabs(w[i][j] - u[i][j]);
         }
      if (diff <= EPSILON) break;
      for (i = 1; i < N-1; i++)
         for (j = 1; j < N-1; j++) u[i][j] = w[i][j];
   }

   /* Print solution */
   for (i = 0; i < N; i++) {
      for (j = 0; j < N; j++)
         printf ("%6.2f ", u[i][j]);
      putchar ('\n');
   }
}
```

Figure 13.10 C program solving the steady-state heat distribution problem using the Jacobi method.

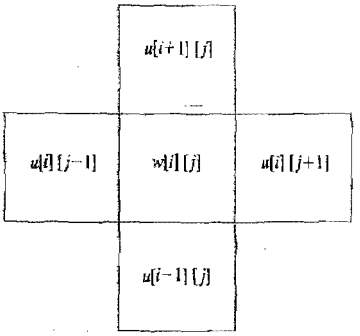


Figure 13.11 The value of `w[i][j]` depends upon the values of `u[i-1][j]`, `u[i][j+1]`, `u[i+1][j]`, and `u[i][j-1]`.

13.4.3 Parallel Program Design

We can associate a primitive task with the computation of each `w[i,j]`. In the Jacobi method the updating step is perfectly parallel. To compute `w[i,j]` each task requires the u values from its neighbors to the north, south, east, and west (Figure 13.11).

We want to agglomerate tasks and assign one agglomerated task to each parallel process. What is the best way to do the agglomeration? If each process is responsible for a rectangular region, then computing elements of w on the interior of the rectangle can be performed using locally available values of u. Computing elements of w on the edge of the rectangle requires values held by other processes.

We can introduce ghost points around each block of values held by a process. After values received from other processes have been stored in the ghost points, then a single doubly nested loop will allow all of the values of w to be computed.

One choice is a rowwise block-striped decomposition (Figure 13.12a). With this decomposition each interior process exchanges messages with two other processes. An obvious alternative is a checkerboard block decomposition. In this case each interior process exchanges messages with four other processes.

13.4.4 Isoefficiency Analysis

Suppose we are working with an $n \times n$ mesh. Since the computation time per mesh point is constant, the computational complexity of the sequential algorithm is $\Theta(n^2)$ per iteration.

Let's consider the rowwise block-striped decomposition. Each of the p processes manages a submesh of size approximately $(n/p) \times n$. During each iteration, every interior process must send n values to each of its neighbors and

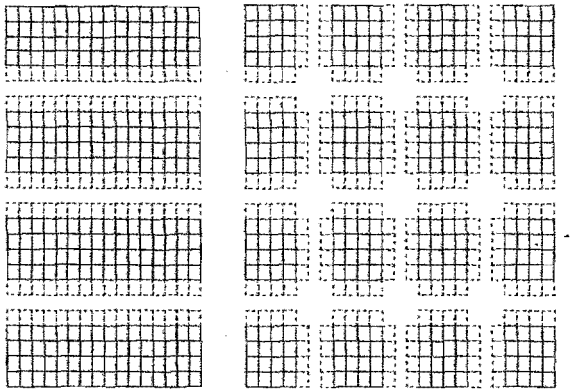


Figure 13.12 Possible data decompositions for solving the two-dimensional steady-state temperature distribution problem. (a) Illustration of a 16 × 16 mesh mapped onto four processes using a rowwise block-striped decomposition. Each process manages an $(n/p) \times n$ region. Ghost points appear as cells edged by dashed lines. (b) Illustration of a 16 × 16 mesh mapped onto 16 processes in a checkerboard block decomposition. Each process manages a region of size $(n/\sqrt{p}) \times (n/\sqrt{p})$. Ghost points are the cells edged by dashed lines.

receive n values from each of them, leading to a communication complexity of $\Theta(n)$. The communication overhead of each iteration of the parallel algorithm is $\Theta(np)$.

The isoefficiency function for the algorithm based on a rowwise block-striped decomposition is

$$n^2 \geq Cnp \Rightarrow n \geq Cp$$

Since $M(n) = n^2$, the scalability function is

$$M(Cp)/p = C^2p^2/p = C^2p$$

The parallel system is not highly scalable.

Now let's look at the checkerboard block-striped decomposition. Each of the p processes manages a submesh of size approximately $(n/\sqrt{p}) \times (n/\sqrt{p})$. During each iteration every interior process must send $n/\sqrt{p}$ values to each of its neighbors and receive $n/\sqrt{p}$ values from them, leading to a communication complexity of $\Theta(n/\sqrt{p})$. The communication overhead of each iteration of the parallel algorithm is $\Theta(n\sqrt{p})$.

The isoefficiency function for the algorithm based on a checkerboard block decomposition is

$$n^2 \geq Cn\sqrt{p} \Rightarrow n \geq C\sqrt{p}$$

Computing the scalability function:

$$M(C\sqrt{p})/p = C^2 p/p = C^2$$

The parallel system is highly scalable.

13.4.5 Implementation Details

Putting ghost points around two-dimensional blocks means message-passing operations require extra copying steps. In C, two-dimensional arrays are stored in row-major order. The ghost points for the top and bottom rows are in contiguous memory locations, but the ghost points for the left and right columns are not. Since the elements are not in contiguous memory locations, you can't receive a message directly into a set of ghost points. Instead, you need to receive messages into a temporary buffer and then copy the values into the ghost points. Similarly, a temporary buffer must be used when assembling values to be sent to a neighboring process's column-oriented ghost points.

13.5 SUMMARY

A partial differential equation is an equation containing derivatives of a function of two or more variables. Scientists and engineers use partial differential equations to model the behavior of a wide variety of physical systems. Realistic problems yield partial differential equations that are too complicated to solve analytically. Instead, scientists and engineers use computers to find approximate solutions to partial differential equations.

The two most common numerical techniques for solving partial differential equations are the finite element method and the finite difference method. Matrix-based implementations of the finite difference method represent the matrix explicitly, using data structures that support efficient access of the nonzero elements. Matrix-free implementations represent the matrix values implicitly. In this chapter we have designed and analyzed parallel programs based on matrix-free implementations of the finite difference method.

Linear second-order partial differential equations can be classified as elliptic, parabolic, and hyperbolic. Different algorithms are used to solve each of these types of PDE, but they do have some similarities. As our case studies, we looked at the solution of the wave equation (an example of a parabolic PDE) and the solution of the heat equation (an example of an elliptical PDE). Hyperbolic PDEs are typically solved by methods not as amenable to parallelization. For each case study, we used our standard parallel algorithm design methodology. We started by identifying primitive tasks and the communication pattern among them. We then chose an agglomeration that represented the best compromise between minimizing communication and maximizing utilization.

In both case studies we used ghost points to store values received from other processes. Once values have been received into the ghost points, all cells can be updated in the same section of code.

We also explored how we could increase the number of ghost points and send extra data values, thereby reducing the frequency of communications at the expense of adding redundant computations. The optimal number of ghost points depends upon latency, bandwidth, and the time needed to compute the value of a cell.

13.6 KEY TERMS

Dirichlet problem
diffusion equation
finite difference method
finite element method
ghost points
Jacobi method
heat equation
Laplace equation
linear second-order partial differential equation
ordinary differential equation
partial differential equation
Poisson equation
wave equation

13.7 BIBLIOGRAPHIC NOTES

Numerical Solution of Partial Differential Equations: Finite Difference Methods, by G. D. Smith, is a detailed examination of finite difference methods [104]. Plybon's book, *An Introduction to Applied Numerical Analysis*, has a chapter on solving partial differential equations using finite difference methods [92]. For a quick introduction to the field, you may find it easier to understand than Smith's monograph.

A comprehensive look at finite element methods is M. J. Fagan's *Finite Element Analysis: Theory and Practice* [23].

Most finite element models and some finite difference models make use of irregular meshes. Much research has been devoted to the problem of decomposing irregular meshes in order to minimize communications and balance computations. *Unstructured Scientific Computation on Scalable Multiprocessors* is an introduction to methods for solving irregular problems on parallel computers [86].

A group centered at Purdue has developed PELLPACK, a problem-solving environment for modeling objects described by partial differential equations. Behind PELLPACK's interactive graphical user interface is more than one million lines of code. For a description of this problem-solving environment, see Houstis et al. [53].

13.8 EXERCISES

13.1 Let $u = f(x, y)$ be a function of two variables. Function u has two unique partial derivatives with order 1, u_x and u_y. Explain why u has only three unique partial derivatives with order 2, rather than four (2×2).

13.2 Write a parallel version of the sequential program that solves the wave equation. Benchmark your program for various values of n and p. Produce a 3-D graph that shows speedup as a function of n and p.

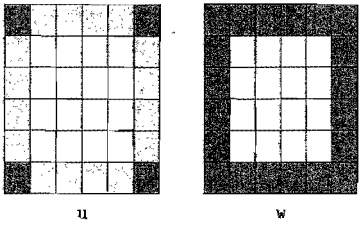


Figure 13.13 A process solving the heat equation is responsible for a 4 × 4 block of the grid. We declare matrices u and w to have 6 rows and 6 columns. The inner white squares contain the values in the process's portion of the grid. The extra rows and columns provide rooms for u's ghost points (gray). The charcoal gray corner elements of matrix u are not used. The inner white squares of w hold its newly computed values; the charcoal gray squares are unused.

13.3 This exercise assumes you are solving the heat equation by assigning a $(n/\sqrt{p}) \times (n/\sqrt{p})$ region to each process. Suppose $k = n/\sqrt{p}$. The matrices u and w on each process have dimensions $(k+2) \times (k+2)$ (see Figure 13.13). The extra rows and columns in u provide room for both elements and ghost points. Making w the same size as u allows us to use the same indices in both matrices to represent the same point in the finite difference mesh.

Write a C code segment that performs the communications needed for processes to update the ghost points of their neighbors.

13.4 Write a parallel version of the sequential program that solves the heat equation. Assume a block-row decomposition of matrices to processes. Benchmark your program for various values of n and p. Produce a 3-D graph that shows speedup as a function of n and p.

13.5 Write a parallel version of the sequential program that solves the heat equation. Assume a two-dimensional block decomposition of matrices to processes. Benchmark your program for various values of n and p. Produce a 3-D graph that shows speedup as a function of n and p.

13.6 Analyze the effectiveness of replicating computations to reduce communications in a parallel program solving the heat equation. Assume each process is responsible for a $(n/\sqrt{p}) \times (n/\sqrt{p})$ block of the mesh.

a. Assume n is an integer multiple of $\sqrt{p}$. Determine the average "communication cost" per iteration, where communication cost includes the time spent performing redundant computations. Your

answer should express average communication cost per iteration as a function of guard wrapper width k, λ, β, and χ.

b. Assume $n = 5000$, $p = 16$, $\chi = 10$ nanosec, $\lambda = 100$ μsec, and $\beta = 5 \times 10^6$ elements/sec. Plot communication cost as a function of ghost wrapper width k, where $1 \leq k \leq 10$.

13.7 Write a program to solve the component labeling problem. A binary image is stored as an $n \times n$ array of 0s and 1s. The 1s represent objects, while the 0s represent empty space between objects. The component labeling problem is to associate a unique positive integer with every object. When the program completes, every 1-pixel will have a positive integer label. A pair of 1-pixels have the same label if and only if they are in the same component (object). The 1-pixels are in the same component if they are linked by a path of 1-pixels. Two 1-pixels are contiguous if they are adjacent to each other, either horizontally or vertically. For example, given this image:

```
1  0  0  0  0  0  0  0
0  1  0  1  0  0  0  0
0  1  1  1  0  0  0  0
0  1  1  0  1  1  1  1
0  0  0  0  1  0  0  1
1  1  1  0  1  1  0  1
1  1  1  1  0  1  1  1
0  0  0  0  0  0  1  1
```

one (but certainly not the only) valid output would be:

```
 1  0  0  0  0  0  0  0
 0 10  0 10  0  0  0  0
 0 10 10 10  0  0  0  0
 0 10 10  0 29 29 29 29
 0  0  0  0 29  0  0 29
41 41 41  0 29 29  0 29
41 41 41 41  0 29 29 29
 0  0  0  0  0  0 29 29
```

Note that a 0 in a particular position of the input image results in a 0 in the same position in the output image. If two positions in the output image have the same integer value, it means there is a path of 1s between the two positions in the input image.

CHAPTER

14

Sorting

Had I been present at the creation, I would have given some useful hints for the better ordering of the universe.

Reaction of Alfonso X to a description of the intricacies of the Ptolemaic system

14.1 INTRODUCTION

Given a sequence of n numbers $\{a_0, a_1, a_2, \ldots, a_{n-1}\}$, the **sorting problem** is to find a permutation $\{a'_0, a'_1, a'_2, \ldots, a'_{n-1}\}$ such that $a'_0 \leq a'_1 \leq a'_2 \leq \ldots \leq a'_{n-1}$. Sorting is one of the most common activities performed on serial computers. Many algorithms incorporate a sort so that information may be accessed efficiently later.

Usually the numbers being sorted are part of data collections called **records.** Within each record, the value being sorted is called the **key.** The rest of the record contains **satellite data.** The information being accessed later is typically in the satellite data, so while it is the keys that are being compared, it is the complete records that must actually be permuted. If there are relatively little satellite data, entire records may be shuffled as the sort progresses. If there are large amounts of satellite data, the sort may actually permute an array of pointers to the records. For the purposes of this chapter, however, we will focus exclusively on the problem of sorting a sequence of numbers, leaving the issues associated with the satellite data as an implementation detail.

Researchers have developed many parallel sorting algorithms. Unfortunately, most of them are designed for theoretical models of parallel computation or special-purpose hardware, making them useless for those trying to implement an efficient sort on a general-purpose parallel computer.

Our focus in this chapter will be on methods suitable for multiple-CPU computers. We'll narrow our coverage in two additional ways. First, we'll be

considering **internal sorts**—algorithms that sort sequences small enough to fit entirely in primary memory. (In contrast, an **external sort** orders a list of values too large to fit at one time in primary memory.) Second, the algorithms we consider here sort by comparing pairs of numbers. (Radix sort is an example of a sort that does not compare pairs of numbers.)

In this chapter we briefly summarize how quicksort works and then develop three parallel quicksort algorithms, assuming our target machine is a modern multicomputer that has equal latency and bandwidth between arbitrary pairs of processors.

14.2 QUICKSORT

Since you're probably familiar with the sequential quicksort algorithm, we'll only present a short refresher here. If you need a more in-depth review, consult Cormen et al. [18], Baase and Van Gelder [5], or another introductory analysis of algorithms textbook.

Quicksort, invented by C. A. R. Hoare about forty years ago [51], is a recursive algorithm that relies upon key comparisons to sort an unordered list. When passed a list of numbers, the algorithm selects one of these numbers to be the pivot. It partitions the list into two sublists: a "low list" containing numbers less than or equal to the pivot, and a "high list" containing those values greater than the pivot. It calls itself recursively to sort the two sublists. (If a sublist has no numbers, the call may be omitted.) The function ends by returning the concatenation of the low list, the pivot, and the high list.

For example, Figure 14.1 illustrates the operation of quicksort as it sorts the list of integers {79, 17, 14, 65, 89, 4, 95, 22, 63, 11}. Let's assume the algorithm always chooses the first list element to be the pivot value. With 79 as the pivot value, the low list contains {17, 14, 65, 4, 22, 63, 11} and the high list contains {89, 95}. The function calls itself recursively for each of these sublists.

The recursive execution of quicksort on the sublist containing {17, 14, 65, 4, 22, 63, 11} begins by removing 17 as the pivot value. The function creates a low list containing {14, 4, 11} and a high list containing {22, 63, 65}. Again, it calls itself recursively for both of these sublists.

The recursion eventually terminates because the removal of the pivot element guarantees the lengths of the lists continue to decrease. If a sublist has no elements, there is no need to sort it. If quicksort is called with a single element, that element becomes the pivot, and the algorithm simply returns that element as the sorted list.

Each function invocation results in the function returning the concatenation of the two sorted sublists and the pivot element. For example, look at the node Q(17,14,65,4,22,63,11). The call to Q(14,4,11) returns {4, 11, 14}. The pivot element is 17. The call to Q(65,22,63) returns {22, 63, 65}. Concatenating these lists, the function returns {4, 11, 14, 17, 22, 63, 65}.

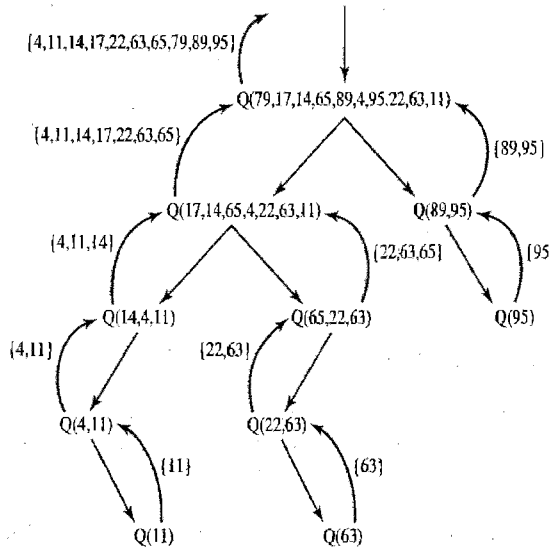


Figure 14.1 Sorting a 10-element list using quicksort. Each Q represents a call to quicksort. The algorithm removes the first element from the list, using it as a pivot to divide the list into two parts. It calls itself recursively to sort the two sublists. (The call is omitted for empty sublists.) It returns the concatenation of the sorted "low list," the pivot, and the sorted "high list."

14.3 A PARALLEL QUICKSORT ALGORITHM

Quicksort is a good starting point for a parallel sorting algorithm for two reasons. First, it is generally recognized as the fastest sorting algorithm based on comparison of keys, in the average case. We always prefer to base our parallel algorithms on the fastest sequential algorithms. Second, quicksort has some natural concurrency. When quicksort calls itself recursively, the two calls may be executed independently.

14.3.1 Definition of Sorted

We want an algorithm suitable for implementation on commodity clusters and multicomputers. Before we go any further, we must determine what it means for a multicomputer to sort an unordered list. We could say that at the beginning of the algorithm a single processor contains the unsorted list in its primary memory, and at the end of the algorithm the same processor would contain the sorted list in its primary memory. The problem with this definition is that it does not allow the maximum problem size to increase with the number of processors.

Instead, we'll adopt a different definition of what we mean by parallel sorting on a multicomputer. We assume that the list of unordered values is initially

distributed evenly among the primary memories of the processors. At the completion of the algorithm, (1) the list stored in every processor's memory is sorted, and (2) the value of the last element on P_i's list is less than or equal to the value of the first element on P_{i+1}'s list, for $0 \leq i \leq p-2$. Note that the sorted values do not need to be distributed evenly among the processors.

14.3.2 Algorithm Development

Let's imagine how a parallel quicksort algorithm could work. Because the quicksort function calls itself twice, the number of "leaves" in the call graph is a power of 2 (ignoring omitted calls due to empty sublists). For this reason we're going to assume that the number of active processes is also a power of 2.

Take a look at Figure 14.2. The unsorted values are distributed among the memories of the processes. We choose a pivot value from one of the processes and broadcast it (Figure 14.2a). Each process divides its unsorted numbers into two lists: those less than or equal to the pivot, and those greater than the pivot. Each process in the upper half of the process list sends its "low list" to a partner process in the lower half of the process list and receives a "high list" in return (Figure 14.2b). Now the processes in the upper half of the process list have values greater than the pivot, and the processes in the lower half of the process list have values less than or equal to the pivot (Figure 14.2c).

At this point the processes divide themselves into two groups and the algorithm recurses. In each process group a pivot value is broadcast (Figure 14.2c). Processes divide their lists and swap values with partner processes (Figure 14.2d).

After $\log p$ recursions, every process has an unsorted list of values completely disjoint from the values held by the other processes. In other words, the largest value held by process i is smaller than the smallest value held by process $i+1$. Each process can sort the list it controls using (what else?) sequential quicksort, and the parallel algorithm terminates.

14.3.3 Analysis

If we were to implement this algorithm, how well would it perform? The execution time of the algorithm begins when the first process starts execution and ends when the last process finishes execution. That is why it is important to make sure all processes have about the same amount of work, so that they will all terminate at about the same time. In this algorithm, the amount of work is related to the number of elements controlled by the process.

Unfortunately, this algorithm is likely to do a poor job of balancing list sizes. For example, take another look at the example of sequential quicksort illustrated in Figure 14.1. The original list-splitting step produces one list of size 7 and another list of size 2. If the pivot value were equal to the median value, we could divide the list into equal parts, but in order to find the median we must go a long way toward sorting the list, which is what we're trying to do in the first place. So it's not practical to insist that the pivot value be the median value.

However, it is clear that we could do a better job balancing the list sizes among the processes if instead of choosing an arbitrary list element to be the

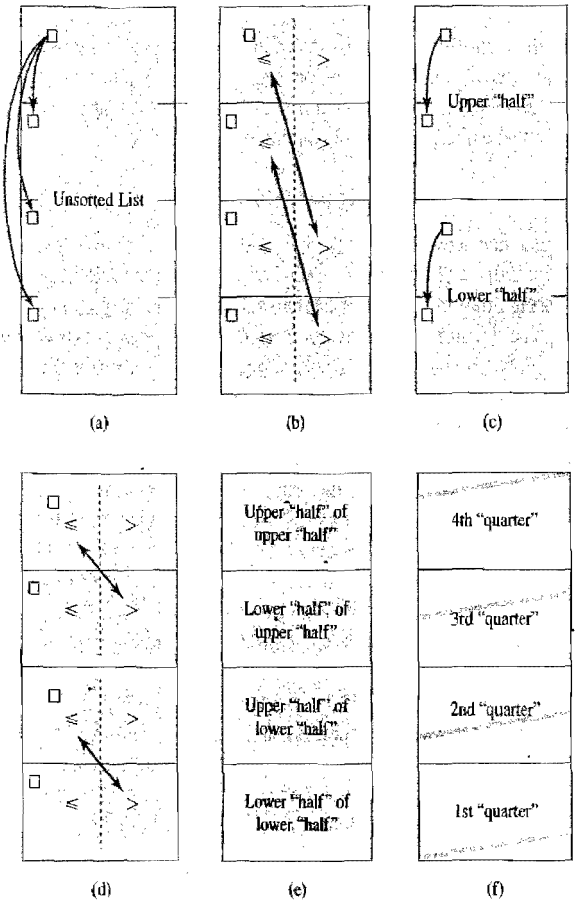


Figure 14.2 High-level view of a parallel quicksort algorithm. (a) Initially the unsorted values are distributed among the memories of all the processes. A single value is chosen as the pivot. The pivot is broadcast to the other processes. (b) Processes use the pivot to divide their numbers into those in the "lower half" and those in the "upper half." Each process in the upper half swaps values with a partner in the lower half. (c) The algorithm recurses. A single value from each "half" is chosen as the pivot for that "half" and broadcast to the other process responsible for that "half." (d) As in step (b), processes use the pivot to divide their numbers. Upper processes swap with lower processes, swapping smaller values for larger values. (e) At this point the largest value held by process i is less than the smallest value held by process $i + 1$. (f) Each process uses quicksort to sort the elements it controls. The list is now sorted.

pivot value, we chose a value more likely to be close to the true median of the sorted list. This insight is the motivation for the next parallel algorithm we will consider: hyperquicksort.

14.4 HYPERQUICKSORT

14.4.1 Algorithm Description

Hyperquicksort, invented by Wagar [108], begins where our first parallel quicksort algorithm ends, with each process using quicksort to sort its portion of the list. At this point condition 1 of the parallel sortedness requirement has been met, but not condition 2.

To meet the second condition, values still need to be moved from process to process. As in the first parallel quicksort algorithm, we will use a pivot value to divide the numbers into two groups: the lower "half" and the upper "half." *Because the list of elements on each process is sorted, the process responsible for supplying the pivot can use the median of its list as the pivot value.* This value is far more likely to be close to the true median of the entire unsorted list than the value of an arbitrarily chosen list element.

The next three steps of hyperquicksort are the same as the parallel quicksort algorithm we already developed. The process choosing the pivot broadcasts it to the other processes. Each process uses the pivot to divide its elements into a "low list" of values less than or equal to the pivot and a "high list" of values greater than the pivot. Every process in the upper half swaps its low list for a high list provided by a partner process in the lower half.

Now we add an additional step to hyperquicksort. After the swap, each process has a sorted sublist it retained and a sorted sublist it received from a partner. It merges the two lists it is responsible for so that the elements it controls are sorted. It is important that processes end this phase with sorted lists, because when the algorithm recurses, two processes will need to choose the median elements of their lists as pivots.

After $\log p$ such split-and-merge steps, the original hypercube of p processes has been divided into $\log p$ single-process hypercubes, and condition 2 is satisfied. Since the processes repeatedly merged lists to keep their local values sorted throughout the divide-and-swap steps, there is no need for them to call quicksort at the end of the algorithm. Figure 14.3 gives an example of hyperquicksort in action.

Hyperquicksort assumes the number of processes is a power of 2. If we arrange the processes as a hypercube, we can set up the communication pattern of the hyperquicksort algorithm so that all communications are between pairs of adjacent processes (see Figure 14.4). For this reason hyperquicksort was a particularly good fit for first-generation multicomputers, such as the Intel iPSC and the nCUBE/ten, that organized processors as a hypercube.

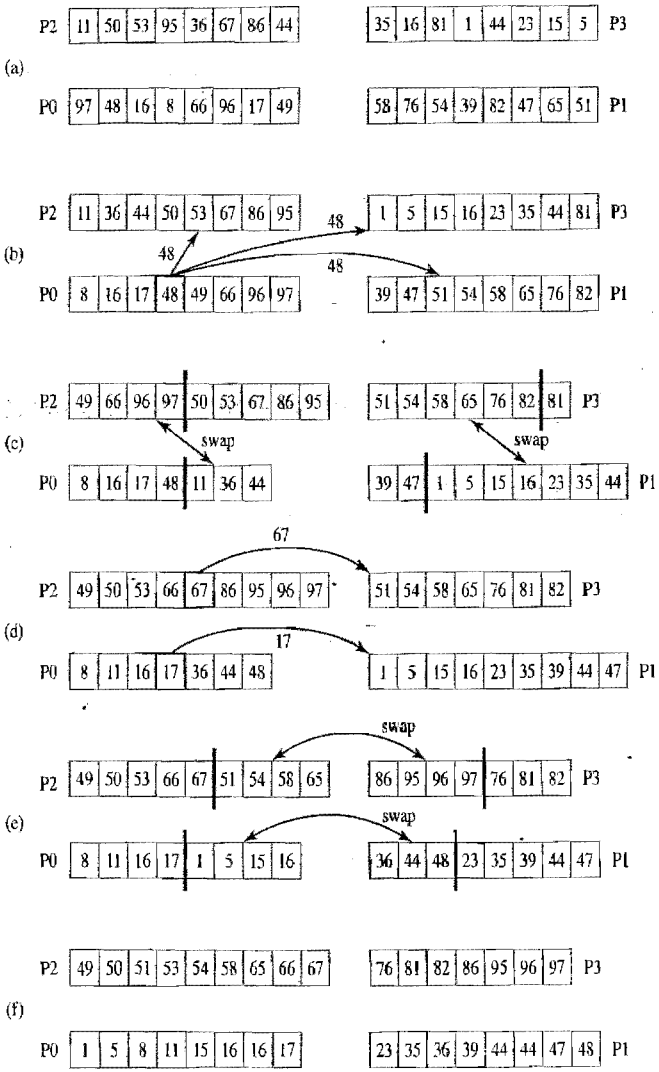


Figure 14.3 Illustration of the hyperquicksort algorithm. In this example 32 elements are being sorted on four processes logically organized as a two-dimensional hypercube. (a) Initially, each process has eight numbers. (b) Each process sorts its own list using quicksort. Process 0 broadcasts its median value, 48, to the other processes. (c) Processes in the lower half of the hypercube send values greater than 48 to processes in the upper half. The processes in the upper half send down values less than or equal to 48. (d) Each process merges the numbers it kept with the numbers it received. Process 0 broadcasts its median value to process 1, and process 2 broadcasts its median value to process 3. (e) Processes swap values across another hypercube dimension. (f) Each process merges the numbers it kept with the numbers it received. At this point the list is sorted.

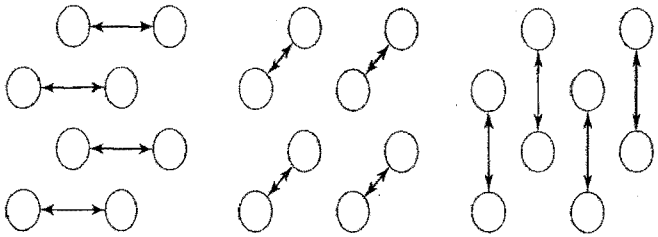


Figure 14.4 Communication pattern of hyperquicksort algorithm. In this example there are eight processes, so the algorithm goes through $\log p = 3$ swap-and-merge steps.

14.4.2 Isoefficiency Analysis

Let's determine the isoefficiency of hyperquicksort. We assume p processors are sorting n elements, where $n \gg p$. At the start of the algorithm each process has no more than $\lceil n/p \rceil$ values. The expected time complexity of the initial quicksort step is $\Theta[(n/p)\log(n/p)]$. Assuming that each process keeps $n/2p$ values and transmits $n/2p$ values in every split-and-merge step, the expected number of comparisons needed to merge the two lists into a single ordered list is n/p. Since the split-and-merge operation is executed for hypercubes of dimension $\log p, (\log p) - 1, \ldots, 1$, the expected number of comparisons performed over the split-and-merge phase of the algorithm is $\Theta(n/p)\log p$, and the expected number of comparisons performed during the entire algorithm is $\Theta[(n/p)(\log n + \log p)]$.

If processes are logically organized as a d-dimensional hypercube, broadcasting the splitter requires communication time $\Theta(d)$. However, since $n \gg p$, the broadcast time will be dwarfed by the time processes spend exchanging list elements. Assuming each process passes half its values each iteration, the time needed to send and receive $n/2p$ sorted values to and from the partner process is $\Theta(n/p)$. There are $\log p$ iterations. Hence the expected communication time for the split-and-merge phase is $\Theta(n \log p/p)$. Since the original quicksort phase requires no interprocess communication, this value is the expected communication complexity of the entire hyperquicksort algorithm.

The sequential time complexity of quicksort is $n \log n$. The communication overhead of hyperquicksort is p times the communication complexity, or $\Theta(n \log p)$. Hence the isoefficiency function for hyperquicksort is

$$n \log n \geq Cn \log p \Rightarrow \log n \geq C \log p \Rightarrow n \geq p^C$$

Memory requirements for this problem are linear; that is, $M(n) = n$. So the scalability function for hyperquicksort is p^{C-1}. The value of C determines the scalability of the parallel system. If $C > 2$, scalability is low.

There is another factor that, when considered, makes the scalability of hyperquicksort even worse. Our analysis has assumed that the median element chosen

by a single process is always the true median, and that every process always sends $n/(2p)$ elements to its partner and receives $n/(2p)$ elements in return each iteration. In reality, the median elements are not the true medians, and the workload among the processes becomes unbalanced. Processes with more elements spend more time communicating and merging. The imbalance tends to increase as the number of processors increases. That's because each process's portion of the complete list is smaller. With a smaller sample, it is less likely that the process's median value will be close to the true median value.

In short, hyperquicksort has two weaknesses that limit its usefulness. First, the expected number of times a key is passed from one process to another is $(\log p)/2$. This communication overhead limits the scalability of the parallel algorithm. We could reduce this overhead if we could find a way to route keys directly to their final destinations. Second, the way in which the splitter values are chosen can lead to workload imbalances among the processes. If we could get samples from all of the processes, we would have a better chance of dividing the list elements evenly among them. These two ideas are incorporated into our third and final algorithm: parallel sorting by regular sampling.

14.5 PARALLEL SORTING BY REGULAR SAMPLING

Parallel sorting by regular sampling (PSRS), developed by Li et al. [74], has three advantages over hyperquicksort. It keeps list sizes more balanced among the processes, it avoids repeated communications of the keys, and it does not require that the number of processes be a power of 2.

14.5.1 Algorithm Description

The PSRS algorithm has four phases (Figure 14.5). Suppose we're sorting n keys on p processes. In phase 1, each process uses the sequential quicksort algorithm to sort its share of the elements (no more than $\lceil n/p \rceil$ elements per process). Each process selects data items at local indices $0, n/p^2, 2n/p^2, \ldots, (p-1)(n/p^2)$ as a regular sample of its locally sorted block.

In the second phase of the algorithm, one process gathers and sorts the local regular samples. It selects $p-1$ pivot values from the sorted list of regular samples. The pivot values are at indices $p + \lfloor p/2 \rfloor - 1, 2p + \lfloor p/2 \rfloor - 1, \ldots, (p-1)p + \lfloor p/2 \rfloor$ in the sorted list of regular samples. At this point each process partitions its sorted sublist into p disjoint pieces, using the pivot values as separators between the pieces.

In the third phase of the algorithm each process i keeps its ith partition and sends the jth partition to process j, for all $j \neq i$.

During the fourth phase of the algorithm each process merges its p partitions into a single list. The values on this list are disjoint from the values on the lists of the other processes. At the end of this phase the elements are sorted.

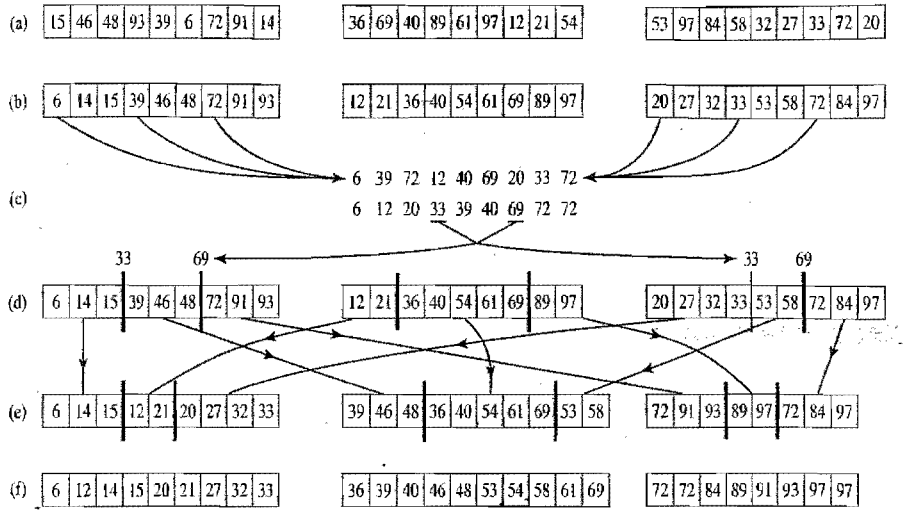


Figure 14.5 This example illustrates how three processes would sort 27 elements using the PSRS algorithm. (a) Original unsorted list of 27 elements is divided among three processes. (b) Each process sorts its share of the list using sequential quicksort. (c) Each process selects regular samples from its sorted sublist. A single process gathers these samples, sorts them, and broadcasts pivot elements from the sorted list of samples to the other processes. (d) Processes use pivot elements computed in step (c) to divide their sorted sublists into three parts. (e) Processes perform an all-to-all communication to migrate the sorted sublist parts to the correct processes. (f) Each process merges its sorted sublists.

Li et al. [74] have proven that the largest number of elements any process may have to merge in phase 4 of the PSRS algorithm is less than $2n/p$; that is, twice its share of the elements. In actuality, experiments have shown that if the elements are selected from a uniform random distribution, the largest partition size is usually no more than a few percent larger than n/p, the average partition size.

14.5.2 Isoefficiency Analysis

Let's determine the isoefficiency of the PSRS algorithm, assuming p processors are sorting n elements, where $n \gg p$.

In phase 1, each process performs quicksort on n/p elements. The time complexity of this step is $\Theta[(n/p)\log(n/p)]$. At the end of phase 1, a single process gathers p regular samples from each of the other $p - 1$ processes. Since relatively few values are being passed, message latency is likely to be the dominant term of this step. Hence the communication complexity of the gather is $\Theta(\log p)$.

In phase 2 of the PSRS algorithm one process sorts the p^2 elements of Y. This sort has time complexity $\Theta(p^2 \log p^2) = \Theta(p^2 \log p)$. The sorting process broadcasts $p - 1$ pivots to the other processes. Since only $p - 1$ values are

being communicated, message latency is most likely the dominant term, and the communication complexity is $\Theta(\log p)$.

In phase 3 of the algorithm, each process uses the pivots to divide its portion of the list into p sections. The processes then perform an all-to-all communication. In the all-to-all communication each process sends and receives $p - 1$ messages. Assuming the list sizes are balanced, the total number of elements sent per process is about $(p - 1)n/p^2$, which is approximately n/p. Since $n \gg p$, the messages are long, and the time needed to pass a message is dominated by the time needed to transmit its elements, rather than its latency. Hence it makes sense to structure the all-to-all communication so that each process sends and receives $p - 1$ messages. That way, every list element is passed only once—directly to the process that needs it. We assume that the processor interconnection network supports p simultaneous message transmissions. In other words, the capacity of the interconnection network increases with the number of processors. (As we saw in Chapter 2, the 4-ary hypertree is an example of an interconnection network for which the bisection width increases linearly with the number of processors.) With this assumption, the overall communication complexity of this step is $\Theta(n/p)$.

In the fourth phase of the algorithm each process merges p sorted sublists. Assuming the list sizes are balanced (which experiments show to be a reasonable assumption), the time required for the merge is $\Theta[(n/p) \log p]$.

The overall computational complexity of the PSRS algorithm is

$$\Theta[(n/p) \log(n/p) + p^2 \log p + (n/p) \log p]$$

Since $n \gg p$, the time needed to sort the regular samples is negligible. The constant of proportionality for the merge step in phase 4 is higher than for the quicksort in phase 1. Hence we need to include the $\Theta[(n/p) \log p]$ term for this phase. Hence the overall computational complexity is

$$\Theta[(n/p)(\log n + \log p)]$$

Assuming the communication capacity of the parallel system increases linearly with p, the overall communication complexity is

$$\Theta(\log p + n/p)$$

Again, since $n \gg p$, the communication time is dominated by the time the processes spend sending sublists to each other, so we can simplify the communication complexity to

$$\Theta(n/p)$$

The parallel overhead of this system is p times the communication complexity, or $\Theta(n)$, plus p times the complexity of the parallel merge step, or $\Theta(n \log p)$. The isoefficiency function for the PSRS algorithm is

$$n \log n \geq Cn \log p \Rightarrow \log n \geq C \log p \Rightarrow n \geq p^C$$

Since $M(n) = n$, the scalability function is

$$p^C/p = p^{C-1}$$

This is the same scalability function we saw for hyperquicksort. However, the PSRS algorithm is likely to achieve higher speedup than hyperquicksort because it keeps the number of keys per processor well balanced.

14.6 SUMMARY

Sorting is an important utility on both serial and parallel computers. In this chapter we have looked at three quicksort-based parallel algorithms suitable for implementation on both multicomputers and multiprocessors.

Our first algorithm introduces the idea of repeatedly halving the lists and exchanging values between pairs of processes until the processes control nonoverlapping sublists. Unfortunately, it does not do a good job balancing values among processes.

Hyperquicksort retains the idea of recursively splitting and exchanging sublists. However, by moving the quicksort step from the end of the algorithm to the beginning, it allows a better choice of the pivot value. The design of hyperquicksort was inspired by the architecture of many 1980s multicomputers having a hypercube processor organization. In these systems the time required to send a message was directly proportional to the number of "hops" between the sending and the receiving processors. Hyperquicksort can be implemented so that all messages are between adjacent processors. Hence it optimizes communication time on hypercubes. Because hyperquicksort relies upon a single process to choose the pivot value for the entire cube (or subcube), as the number of processors grows, the quality of the pivot value degrades. As the pivot value strays from the true median, the workloads among the processes become imbalanced, lowering efficiency.

Parallel sorting by regular sampling (PSRS) addresses the load imbalance problem of hyperquicksort by choosing pivot elements from a regular sample of elements held by all the processes. It has the additional advantage that its single all-to-all communication can be implemented so that each element is moved only once (rather than $\log p$ times). This is a good fit for contemporary switch-based clusters, in which the time needed to send a message is about the same for any pair of processors.

14.7 KEY TERMS

external sort
hyperquicksort
internal sort
key
parallel sorting by regular sampling
record
satellite data
sorting problem

14.8 BIBLIOGRAPHIC NOTES

Parallel sorting algorithms have been the object of much study. In fact, an entire book has been devoted to the topic: *Parallel Sorting Algorithms* by Akl [1].

In 1968 Batcher introduced a parallel sorting algorithm called bitonic merge [8]. A network of $n/2$ comparator elements can sort a list of n values in time $\Theta(\log^2 n)$. However, algorithms based on bitonic merge have poor scalability on multiprocessors and multicomputers [44]. For this reason, we have not discussed parallel implementations of bitonic mergesort in this chapter.

Various authors have proposed multiprocessor sorting algorithms that have two phases: a phase in which each process quicksorts its own subset of the data, followed by a phase in which processes cooperate to merge their sorted subsets. References to such algorithms in the literature include work by Francis and Mathieson [35], Quinn [94], and Wheat and Evans [113].

Quinn [94] discusses a parallel shell sort on UMA multiprocessors. Fox et al. [33] and Grama et al. [44] also describe a parallel "shell sort" algorithm that is not a strict parallelization of the sequential algorithm but has the same flavor.

14.9 EXERCISES

14.1 A *stable* sorting algorithm preserves the original order of keys with the same value. Is quicksort a stable sorting algorithm?

14.2 Assume we are sorting four-byte keys on a multicomputer with 16 processors. Assume it takes 70 nanoseconds to compare two keys, message latency is 200 μsec, and message bandwidth is 10^7 byte/sec.

a. Perform a computational experiment to determine the expected maximum list size held by any process after the $\log p$ list-splitting steps of the first parallel quicksort algorithm, for $1 \leq p \leq 16$. In each case compare the expected maximum list size to $\lceil n/p \rceil$.

b. Predict the speedup achievable by this parallel computer using the first parallel quicksort algorithm to sort 100 million keys on $1, 2, \ldots, 16$ processors.

14.3 Assume we are sorting four-byte keys on a multicomputer with 16 processors. Assume it takes 70 nanoseconds to compare two keys, message latency is 200 μsec, and message bandwidth is 10^7 byte/sec.

a. Perform a computational experiment to determine the expected maximum list size held by any process after the $\log p$ list-splitting steps of the hyperquicksort algorithm, for $1 \leq p \leq 16$. In each case compare the expected maximum list size to $\lceil n/p \rceil$.

b. Predict the speedup achievable by this parallel computer using the hyperquicksort algorithm to sort 100 million keys on $1, 2, \ldots, 16$ processors.

14.4 In the average case about $1.386\, n \log n - 2.846\, n$ comparisons are performed by sequential quicksort sorting n keys [5]. Assume we are sorting four-byte keys on a multicomputer. Assume it takes 70 nanoseconds to compare two keys, message latency is 200 μsec, and message bandwidth is 10^7 byte/sec. The PSRS algorithm typically divides the keys nearly evenly, so that at the final step no process is responsible for merging more than about $\lceil 1.03\, n/p \rceil$ keys. Predict the speedup achievable by this parallel computer using the PSRS algorithm to sort 100 million keys on $1, 2, \ldots, 16$ processors.

14.5 Our analyses of hyperquicksort and the PSRS algorithm assume that the initial ordering of the n keys is a random permutation of their sorted order; that is, each of the $n!$ possible permutations is equally likely. Suppose the initial list of keys is already sorted; that is, $a_0 \leq a_1 \leq \ldots \leq a_{n-1}$. Which of the two algorithms is less disrupted by this permutation? Why?

14.6 In the last step of the PSRS algorithm, each process must merge p sorted sublists, each of size about n/p^2. (The total number of elements to be merged is typically about n/p and guaranteed to be less than $2n/p$.) Describe the algorithm(s) and data structure(s) that can accomplish the merge in time $\Theta[(n/p) \log p]$.

14.7 a. Write a parallel program implementing the hyperquicksort algorithm.

b. Using your system's values for χ, λ, and β, predict the speedup your program will achieve for various numbers of processors p and various problem sizes n.

c. Benchmark the program for the same combinations of p and n.

d. What is the error between the prediction and the experimental results? Identify the largest source of error in your speedup formula.

14.8 a. Write a parallel program implementing the PSRS algorithm.

b. Using your system's values for χ, λ, and β, predict the speedup your program will achieve for various numbers of processors p and various problem sizes n.

c. Benchmark the program for various combinations of p and n.

d. What is the error between the prediction and the experimental results? Identify the largest source of error in your speedup formula.

14.9 This chapter has focused on parallel implementations of quicksort, a $\Theta(n \log n)$ sorting algorithm. Another well-known, recursive, $\Theta(n \log n)$ sorting algorithm is mergesort. Here is pseudocode for mergesort. It relies on function Merge, which merges two sorted lists.

```
Mergesort (list):
  if length(list) = 1 then
    return list
  else
    part1 ← Mergesort (first half of list)
    part2 ← Mergesort (remainder of list)
    return Merge (part1, part2)
  endif
```

If you need a more detailed explanation of mergesort, consult Cormen et al. [18], Baase and Van Gelder [5], or another introductory analysis of algorithms textbook.

a. Design a parallel version of mergesort for a multicomputer. Make these three assumptions. At the beginning of the algorithm's execution all unsorted values are in the memory of one processor. At the end of the algorithm's execution the sorted list is in the memory of one processor. The number of processors p is a power of 2.

b. What is the complexity of this algorithm?

c. What is the isoefficiency of your parallel mergesort algorithm?

d. Write a program implementing your parallel algorithm. Benchmark the program for various combinations of p and n.

14.10 Suppose the values of n keys are uniformly distributed in the interval $[0, k)$. *Bucket sort* divides the interval $[0, k)$ into j equal-sized subintervals called *buckets*. Each key is placed in one of the buckets, based upon its value. After all the keys have been placed in buckets, the keys in each bucket are sorted. Once the keys in each bucket are sorted, the sequence is sorted.

a. Design a parallel version of bucket sort. Initially the n keys are distributed among the p processes. The interval $[0, k)$ should be divided into p buckets. In step 1, every process divides its n/p keys into p groups, one per bucket. In step 2, each process assumes responsibility for one of the buckets. An all-to-all communication routes groups of keys to the correct processes. In step 3, each process sorts the keys in its bucket.

b. What is the complexity of this algorithm?

c. What is the isoefficiency of your parallel bucket sort algorithm?

d. Write a program implementing parallel bucket sort. Benchmark the program for various combinations of p and n.

14.11 Write a parallel program to find the kth largest element in an unsorted list of n elements initially distributed among p processors.

14.12 A file contains n signed integers, each four bytes long. Write a parallel program to determine which integer value occurs most frequently in the file.

CHAPTER

15

The Fast Fourier Transform

The meeting of two personalities is like the contact of two chemical substances; if there is any reaction, both are transformed.

Carl Gustav Jung, *Modern Man in Search of a Soul*

15.1 INTRODUCTION

The discrete Fourier transform has many applications in science and engineering. For example, it is often used in digital signal processing applications such as voice recognition and image processing. A straightforward implementation of the discrete Fourier transform has time complexity $\Theta(n^2)$. The fast Fourier transform is a $\Theta(n \log n)$ algorithm to perform the discrete Fourier transform, and it can be parallelized easily.

In this chapter we illuminate how the discrete Fourier transform works by using an example from speech recognition. We formally present the discrete Fourier transform and the inverse discrete Fourier transform. We move on to present the fast Fourier transform algorithm and describe how to implement it on a multicomputer.

15.2 FOURIER ANALYSIS

Fourier analysis studies the representation of continuous functions by a potentially infinite series of sinusoidal (sine and cosine) functions. We can view the discrete Fourier transform as a function that maps a sequence over time $\{f(k)\}$ to another sequence over frequency $\{F(j)\}$. The sequence $\{f(k)\}$ represents a sampling of a signal's distribution as a function of time. The sequence

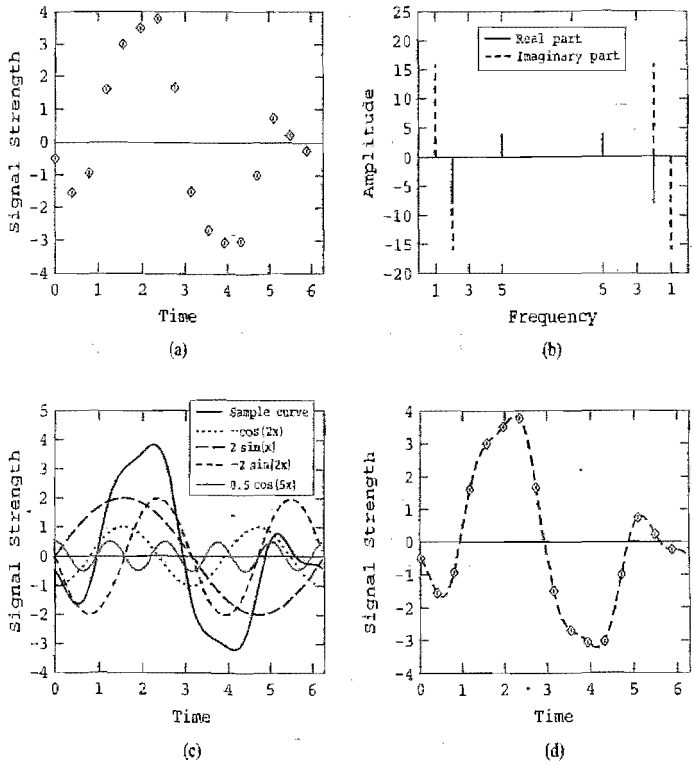


Figure 15.1 Example of the discrete Fourier transform. (a) A set of 16 data points representing samples of signal strength in the time interval 0 to (but not through) 2π. (b) The discrete Fourier transform yields the amplitudes and frequencies of the constituent sine and cosine functions. (c) A plot of the four constituent functions and their sum, a continuous function. (d) A plot of the continuous function and the original 16 samples.

$\{F(j)\}$ represents a distribution of Fourier coefficients as a function of frequency. We can use $\{F(j)\}$ to compute the sinusoidal components of the sampled signal.

Figure 15.1 illustrates this process. We begin, in Figure 15.1a, with a plot of $\{f(k)\}$, 16 samples of signal strength between time 0 and time 2π. Figure 15.1b is the plot of $\{F(j)\}$, a sequence of 16 complex numbers representing the frequency distribution. From the nonzero elements of $\{F(j)\}$ we can determine the frequency of the terms generating the signal, where **frequency** means the number of complete cycles the wave completes between time 0 and time 2π. Nonzero real components correspond to cosine functions; nonzero imaginary components correspond to sine functions. From Figure 15.1b we see that there are nonzero real components with frequency 2 and 5 and nonzero imaginary components with

frequency 1 and 2. Hence the function generating the signal is of the form

$$s_1 \sin x + c_2 \cos(2x) + s_2 \sin(2x) + c_5 \cos(5x)$$

For each frequency, we divide the amplitude shown in the left half of Figure 15.1b by 8 (half of 16, the number of sample points) to determine the coefficients of the various sinusoidal components. The frequency 1 component is $16i$. Dividing 16 by 8 yields a coefficient of 2 for the function $\sin x$. The frequency 2 component is $-8 - 16i$. Dividing -8 by 8 yields a coefficient of -1 for the function $\cos(2x)$, just as the coefficient for the function $\sin(2x)$ is -2. We use the same method to calculate that 0.5 is the coefficient of the function $\cos(5x)$. The four terms generating the signal are

$$2 \sin x - \cos(2x) - 2 \sin(2x) + 0.5 \cos(5x)$$

In Figure 15.1c we plot the four sinusoidal components and their sum, a continuous function, and in Figure 15.1d we plot the continuous function against the sampled data points.

Let's look at how Fourier analysis is used in speech recognition. Most speech analysis has been done by studying the spectral parameters of the speech signal.

> Decomposing complex speech signals into periodically recurrent sinusoidal components is the central activity of signal processing work, and is justified by (1) sinusoids being "natural signals" of linear physical (electronic) systems; (2) resonances being prominent cues to articulation configurations; (3) voice sounds being composed out of harmonics of the voice fundamental frequency; and (4) the ear appearing to do some form of spectral analysis. Also, sinusoids (and some other exponential signals) can be added ("superimposed") in linear systems without interfering with each other; thus the sinusoidal parts that we decompose the signal input into for frequency analysis act as independent, "orthogonal signals" [67].

The discrete Fourier transform can be used to convert digitized samples of human speech into two-dimensional plots (see Figure 15.2). The graph shows detected frequencies as a function of time. Each narrow vertical strip shows the amplitudes of the detected frequencies as shades of gray. As the person talks, the speech signal changes, and so do the frequencies that make up the signal. Plots such as this can be used as inputs to speech recognition systems, which try to identify spoken phonemes through pattern recognition.

15.3 THE DISCRETE FOURIER TRANSFORM

Given an n element vector x, the **discrete Fourier transform (DFT)** is the matrix-vector product $F_n x$, where $f_{i,j} = \omega_n^{ij}$ for $0 \leq i, j < n$ and ω_n is the primitive nth root of unity. (For a review of complex numbers, refer to Appendix D.)

For example, to compute the discrete Fourier transform of the vector (2, 3), we need to know ω_2, the primitive square root of unity. The primitive square root

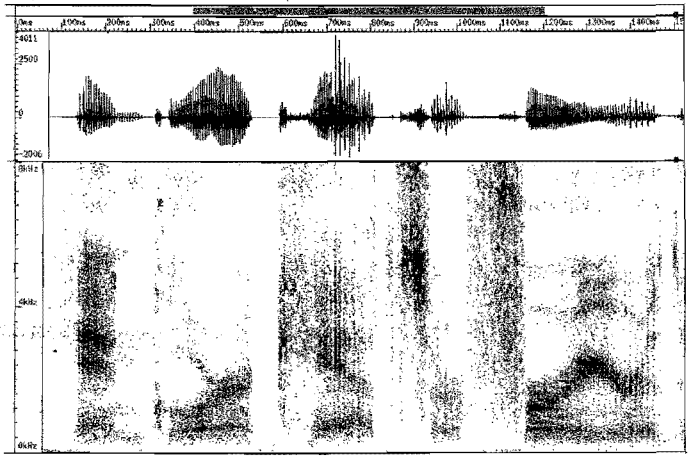


Figure 15.2 Discrete Fourier transform of the waveform corresponding to "Angora cats are furrier...." The upper portion of the chart plots the strength of the input signal as a function of time. The lower portion plots frequency and amplitude as a function of time. Each narrow vertical strip represents the discrete Fourier transform of the waveform using a moving 10 ms window within a 3 ms increment. The darker the plot at some vertical position, the higher the amplitude at that frequency. (Figure courtesy Ron Cole and Yeshwant Muthusamy of the Oregon Graduate Institute.)

of unity is -1. The DFT of (2, 3), then, is

$$\begin{pmatrix} \omega_2^{0\times 0} & \omega_2^{0\times 1} \\ \omega_2^{1\times 0} & \omega_2^{1\times 1} \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \begin{pmatrix} 2 \\ 3 \end{pmatrix} = \begin{pmatrix} 5 \\ -1 \end{pmatrix}$$

Now let's compute the DFT of the vector (1, 2, 4, 3). We will need to use the primitive fourth root of unity, which is i.

$$\begin{pmatrix} \omega_4^0 & \omega_4^0 & \omega_4^0 & \omega_4^0 \\ \omega_4^0 & \omega_4^1 & \omega_4^2 & \omega_4^3 \\ \omega_4^0 & \omega_4^2 & \omega_4^4 & \omega_4^6 \\ \omega_4^0 & \omega_4^3 & \omega_4^6 & \omega_4^9 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & i & -1 & -i \\ 1 & -1 & 1 & -1 \\ 1 & -i & -1 & i \end{pmatrix} \begin{pmatrix} 1 \\ 2 \\ 4 \\ 3 \end{pmatrix} = \begin{pmatrix} 10 \\ -3-i \\ 0 \\ -3+i \end{pmatrix}$$

Let's put the DFT to use by returning to the example presented in the previous section. We have a vector of 16 complex numbers representing signal strength in the time interval 0 to 2π. To simplify the presentation we show each number to only three digits of accuracy:

$$(-0.500, -1.55, -0.939, 1.60, 3.00, 3.51, 3.77, 1.66,$$
$$-1.50, -2.70, -3.06, -3.02, -1.00, 0.736, 0.232, -0.250)$$

The DFT of this vector is

$$(0, 16i, -8-16i, 0, 0, 4, 0, 0, 0, 0, 0, 4, 0, 0, -8+16i, -16i)$$

To determine the coefficients of the sine and cosine functions making up this signal, we examine the nonzero elements in the first half of the transformed sequence. (The terms at positions 9 through 15 are a reflection of the terms in positions 1 through 7, with the signs of the imaginary parts reversed.) If we begin counting at 0, the real portion of term k is 8 times the coefficient of the function $\cos(kx)$, and the imaginary portion of term k is 8 times the coefficient of the function $\sin(kx)$. (Eight is half the number of sample points.) Thus the combination of sine and cosine functions making up the curve is

$$2\sin(x) - \cos(2x) - 2\sin(2x) + 0.5\cos(5x)$$

15.3.1 Inverse Discrete Fourier Transform

Given an n element vector x, the **inverse discrete Fourier transform (inverse DFT)** is $1/n$th the matrix-vecror product $F_n^{-1}x$, where $f^{-1}(ij) = \omega_n^{-ij}$ for $0 \leq i, j < n$ and ω_n is the primitive n root of unity.

For example, the inverse DFT of the vector $(10, -3-i, 0, -3+i)$ is

$$\frac{1}{4}\begin{pmatrix} \omega_4^0 & \omega_4^0 & \omega_4^0 & \omega_4^0 \\ \omega_4^0 & \omega_4^{-1} & \omega_4^{-2} & \omega_4^{-3} \\ \omega_4^0 & \omega_4^{-2} & \omega_4^{-4} & \omega_4^{-6} \\ \omega_4^0 & \omega_4^{-3} & \omega_4^{-6} & \omega_4^{-9} \end{pmatrix}\begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix} = \frac{1}{4}\begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & -i & -1 & i \\ 1 & -1 & 1 & -1 \\ 1 & i & -1 & -i \end{pmatrix}\begin{pmatrix} 10 \\ -3-i \\ 0 \\ -3+i \end{pmatrix}$$

$$= \frac{1}{4}\begin{pmatrix} 4 \\ 8 \\ 16 \\ 12 \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \\ 4 \\ 3 \end{pmatrix}$$

15.3.2 Sample Application: Polynomial Multiplication

We can use the DFT and inverse DFT to multiply polynomials. First, we need to understand what the DFT and inverse DFT do. The DFT evaluates a polynomial at the n complex nth roots of unity. Let's see why this is true. If $f(x) = a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + \cdots + a_1x + a_0$ is a polynomial of degree $n-1$, and ω is the primitive nth root of unity, then

$$\begin{pmatrix} f(\omega^0) \\ f(\omega^1) \\ \cdots \\ f(\omega^{n-1}) \end{pmatrix} = F\begin{pmatrix} a_0 \\ a_1 \\ \cdots \\ a_{n-1} \end{pmatrix}$$

because $f(\omega^i) = a_0 + a_1\omega^i + a_2\omega^{2i} + \cdots + a_{n-1}\omega^{(n-1)i}$ for $0 \leq i < n$.

The inverse DFT takes the values of a polynomial at the n complex nth roots of unity and produces the polynomial's coefficients.

Now, suppose we want to multiply two polynomials

$$p(x) = \sum_{i=0}^{n-1} a_i x^i \quad \text{and} \quad q(x) = \sum_{i=0}^{n-1} b_i x^i$$

The product of these two polynomials of degree $n - 1$ is the $(2n - 2)$ degree polynomial

$$p(x)q(x) = \sum_{i=0}^{2n-2} \sum_{j=0}^{i} a_j b_{i-j} x^i$$

We can compute the coefficients of the resulting polynomial $p(x)q(x)$ by convoluting the coefficient vectors of the original polynomials.

For example, to multiply the two polynomials

$$p(x) = 2x^3 - 4x^2 + 5x - 1$$
$$q(x) = x^3 + 2x^2 + 3x + 2$$

yielding

$$r(x) = a_6x^6 + a_5x^5 + a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0$$

we convolute the coefficient vectors:

$$a_6 = 2 \times 1 = 2$$
$$a_5 = 2 \times 2 + (-4) \times 1 = 0$$
$$a_4 = 2 \times 3 + (-4) \times 2 + 5 \times 1 = 3$$
$$a_3 = 2 \times 2 + (-4) \times 3 + 5 \times 2 + (-1) \times 1 = 1$$
$$a_2 = (-4) \times 2 + 5 \times 3 + (-1) \times 2 = 5$$
$$a_1 = 5 \times 2 + (-1) \times 3 = 7$$
$$a_0 = (-1) \times 2 = -2$$

resulting in

$$r(x) = 2x^6 + 3x^4 + x^3 + 5x^2 + 7x - 2$$

Another way to multiply two polynomials of degree $n - 1$ is to evaluate them at the n complex nth roots of unity, perform an element-wise multiplication of the polynomials' values at these points, and then interpolate the results to produce the coefficients of the product polynomial. Let's apply this method to the previous example.

First we perform the DFT on the coefficients of $p(x)$. We list the coefficients in order from low to high. Since the polynomial has degree 3, the last four coefficients

are 0. To simplify the figure, we only show two digits past the decimal point.

$$\begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & \omega^1 & \omega^2 & \omega^3 & \omega^4 & \omega^5 & \omega^6 & \omega^7 \\ 1 & \omega^2 & \omega^4 & \omega^6 & \omega^0 & \omega^2 & \omega^4 & \omega^6 \\ 1 & \omega^3 & \omega^6 & \omega^1 & \omega^4 & \omega^7 & \omega^2 & \omega^5 \\ 1 & \omega^4 & 1 & \omega^4 & 1 & \omega^4 & 1 & \omega^4 \\ 1 & \omega^5 & \omega^2 & \omega^7 & \omega^4 & \omega^1 & \omega^6 & \omega^3 \\ 1 & \omega^6 & \omega^4 & \omega^2 & 1 & \omega^6 & \omega^4 & \omega^2 \\ 1 & \omega^7 & \omega^6 & \omega^5 & \omega^4 & \omega^3 & \omega^2 & \omega^1 \end{pmatrix} \begin{pmatrix} 1 \\ 5 \\ -4 \\ 2 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 2 \\ 1.12 + .95i \\ 3 + 3i \\ -3.12 + 8.95i \\ -12 \\ -3.12 - 8.95i \\ 3 - 3i \\ 1.12 - .95i \end{pmatrix}$$

Next we perform the DFT on the coefficients of $q(x)$. Again, we are only showing two digits beyond the decimal point.

$$\begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & \omega^1 & \omega^2 & \omega^3 & \omega^4 & \omega^5 & \omega^6 & \omega^7 \\ 1 & \omega^2 & \omega^4 & \omega^6 & \omega^0 & \omega^2 & \omega^4 & \omega^6 \\ 1 & \omega^3 & \omega^6 & \omega^1 & \omega^4 & \omega^7 & \omega^2 & \omega^5 \\ 1 & \omega^4 & \omega^1 & \omega^4 & 1 & \omega^4 & \omega^1 & \omega^4 \\ 1 & \omega^5 & \omega^2 & \omega^7 & \omega^4 & \omega^1 & \omega^6 & \omega^3 \\ 1 & \omega^6 & \omega^4 & \omega^2 & 1 & \omega^6 & \omega^4 & \omega^2 \\ 1 & \omega^7 & \omega^6 & \omega^5 & \omega^4 & \omega^3 & \omega^2 & \omega^1 \end{pmatrix} \begin{pmatrix} 2 \\ 3 \\ 2 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 8 \\ 3.41 + 4.83i \\ 2i \\ .59 + .83i \\ 0 \\ .59 - .83i \\ -2i \\ 3.41 - 4.83i \end{pmatrix}$$

Now we perform an element-wise multiplication of the two polynomials at these eight points.

$$\begin{pmatrix} 2 \\ 1.12 + .95i \\ 3 + 3i \\ -3.12 + 8.95i \\ -12 \\ -3.12 - 8.95i \\ 3 - 3i \\ 1.12 - .95i \end{pmatrix} \begin{pmatrix} 8 \\ 3.41 + 4.83i \\ 2i \\ .59 + .83i \\ 0 \\ .59 - .83i \\ -2i \\ 3.41 - 4.83i \end{pmatrix} = \begin{pmatrix} 16 \\ -.76 + 8.66i \\ -6 + 6i \\ -9.25 + 2.66i \\ 0 \\ -9.25 - 2.66i \\ -6 - 6i \\ -.76 - 8.66i \end{pmatrix}$$

In the final step we perform the inverse DFT on the product vector. Note that we have replaced the negative powers of ω with equivalent values expressed as positive powers. For example, when ω is the primitive 8th root of unity, $\omega^{-1} = \omega^7$ and $\omega^{-2} = \omega^6$. Here is the inverse DFT:

$$\frac{1}{8}\begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & \omega^7 & \omega^6 & \omega^5 & \omega^4 & \omega^3 & \omega^2 & \omega^1 \\ 1 & \omega^6 & \omega^4 & \omega^2 & 1 & \omega^6 & \omega^4 & \omega^2 \\ 1 & \omega^5 & \omega^2 & \omega^7 & \omega^4 & \omega^1 & \omega^6 & \omega^3 \\ 1 & \omega^4 & 1 & \omega^4 & 1 & \omega^4 & 1 & \omega^4 \\ 1 & \omega^3 & \omega^6 & \omega^1 & \omega^4 & \omega^7 & \omega^2 & \omega^5 \\ 1 & \omega^2 & \omega^4 & \omega^6 & 1 & \omega^2 & \omega^4 & \omega^6 \\ 1 & \omega^1 & \omega^2 & \omega^3 & \omega^4 & \omega^5 & \omega^6 & \omega^7 \end{pmatrix} \begin{pmatrix} 16 \\ -.76 + 8.66i \\ -6 + 6i \\ -9.25 + 2.66i \\ 0 \\ -9.25 - 2.66i \\ -6 - 6i \\ -.76 - 8.66i \end{pmatrix} = \begin{pmatrix} -2 \\ 7 \\ 5 \\ 1 \\ 3 \\ 0 \\ 2 \\ 0 \end{pmatrix}$$

The vector produced by the inverse DFT contains the coeffients of the product polynomial in order from low to high. In other words,

$$r(x) = 2x^6 + 3x^4 + x^3 + 5x^2 + 7x - 2$$

15.4 THE FAST FOURIER TRANSFORM

At the end of Section 15.3 we demonstrated how we can use the DFT and inverse DFT to multiply two polynomials. Why would we use this complicated algorithm to perform convolutions or multiply polynomials, when these can be done directly in time $\Theta(n^2)$? The reason is that we do not have to perform the DFT and inverse DFT using matrix-vector multiplication. An algorithm with complexity $\Theta(n \log n)$ exists, and (luckily for us) it is amenable to parallelization. The improved algorithm is called the **fast Fourier transform (FFT).**

The FFT uses a divide-and-conquer strategy to evaluate a polynomial of degree n at the n complex nth roots of unity. To evaluate $f(x)$, a polynomial of degree n where n is a power of 2, the algorithm defines two new polynomials of degree $n/2$. Function $f^{[0]}(x)$ contains the elements of $f(x)$ associated with the even powers of x, while function $f^{[1]}(x)$ contains the elements associated with the odd powers of x:

$$f^{[0]} = a_0 + a_2 x + a_4 x^2 + \cdots + a_{n-2} x^{n/2-1}$$
$$f^{[1]} = a_1 + a_3 x + a_5 x^2 + \cdots + a_{n-1} x^{n/2-1}$$

Note that $f(x) = f^{[0]}(x^2) + x f^{[1]}(x^2)$, so the problem of evaluating $f(x)$ at the points $\omega_n^0, \omega_n^1, \ldots, \omega_n^{n-1}$ reduces to evaluating $f^{[0]}$ and $f^{[1]}$ at $(\omega_n^0)^2$, $(\omega_n^1)^2, \ldots, (\omega_n^{n/2-1})^2$, and then computing $f(x) = f^{[0]}(x^2) + x f^{[1]}(x^2)$.

Halving Lemma If n is an even positive number, then the squares of the n complex nth roots of units are identical to the $n/2$ complex $(n/2)$th roots of unity.

Proof See Appendix D.

By the **halving lemma,** we know that the set of points $(\omega_n^0)^2, (\omega_n^1)^2, \ldots, (\omega_n^{n-1})^2$ consists of only $n/2$ unique values. In other words, to evaluate the polynomial $f(x)$ at the n complex nth roots of unity we need only evaluate the polynomials $f^{[0]}(x)$ and $f^{[1]}(x)$ at the $n/2$ complex $(n/2)$th roots of unity. Hence our divide-and-conquer strategy will save us computations.

The most natural way to express the FFT algorithm resulting from the divide-and-conquer strategy is to use recursion. Pseudocode for a recursive implementation of FFT appears in Figure 15.3. The time complexity of this algorithm is easy to determine. Let $T(n)$ denote the time needed to perform the FFT on a polynomial of degree n, where n is a power of 2.

$$\begin{aligned} T(n) &= 2T(n/2) + \Theta(n) \\ &= \Theta(n \log n) \end{aligned}$$

```
Recursive_FFT (a, n)

  Parameter n              Number of elements in a
            a[0...(n - 1)] Coefficients
  Local     ω_n            Primitive nth root of unity
            ω              Evaluate polynomial at this point
            a^[0]          Even-numbered coefficients
            a^[1]          Odd-numbered coefficients
            y              Result of transform
            y^[0]          Result of FFT of a^[0]
            y^[1]          Result of FFT of a^[1]

if n = 1 then return a
else
  ω_n ← e^(2πi/n)
  ω ← 1
  a^[0] ← (a[0], a[2], ..., a[n - 2])
  a^[1] ← (a[1], a[3], ..., a[n - 1])
  y^[0] ← Recursive_FFT (a^[0], n/2)
  y^[1] ← Recursive_FFT (a^[1], n/2)
  for k ← 0 to n/2 - 1 do
    y[k] ← y^[0][k] + ω × y^[1][k]
    y[k + n/2] ← y^[0][k] - ω × y^[1][k]
    ω ← ω × ω_n
  endfor
  return y
endif
```

Figure 15.3 Recursive sequential implementation of the fast Fourier transform algorithm (adapted from Cormen et al. [18]).

While the recursive formulation of the FFT algorithm is (relatively) easy to understand, we have two reasons for developing an iterative FFT algorithm. First, a well-written iterative version of the FFT algorithm can perform fewer index computations and eliminate the second evaluation of $\omega_n^k y^{[1]}[k]$ every iteration of the for loop. Second, it is easier to derive a parallel FFT algorithm when the sequential algorithm is in iterative form.

Figure 15.4 illustrates the derivation of an iterative algorithm from the recursive algorithm. In Figure 15.4a we see how the recursive algorithm transforms a vector of four elements. Each rounded rectangle represents a call to function `fft`. The vector to be transformed is inside the parentheses. The function keeps dividing the vector in half and calling itself recursively on each half until the vector size is 1. The DFT of a single value is that value. (Remember the FFT is simply a fast way of performing the DFT.) The heavy curved arrows show the values returned from each invocation of the function. The function combines the values received in two vectors of length i and returns a vector of length $2i$. Performing the FFT on input vector $(1, 2, 4, 3)$ produces the result vector $(10, -3 - i, 0, -3 + i)$.

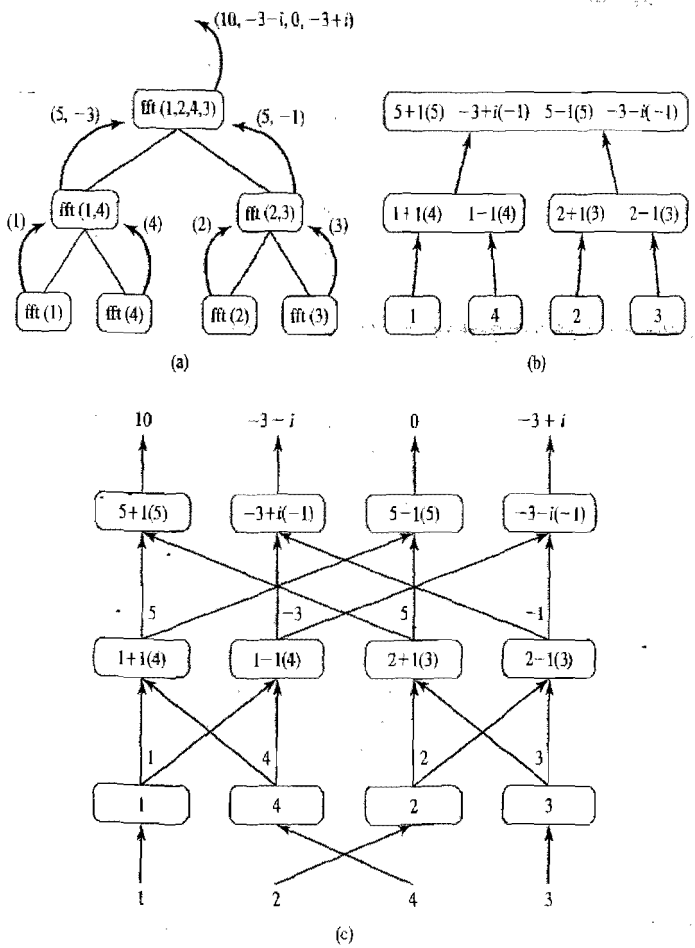


Figure 15.4 Evolution of the iterative algorithm from the recursive algorithm. (a) Recursive implementation of FFT. (b) Determining which computations are performed for each function invocation. (c) Tracking the flow of data values.

In Figure 15.4b we look inside the functions and determine exactly which operations are performed for each invocation. The expressions of form $a + b(c)$ and $a - b(c)$ correspond to the pseudocode statements

$$y[k] \leftarrow y^{[0]}[k] + \omega \times y^{[1]}[k]$$
$$y[k + n/2] \leftarrow y^{[0]}[k] - \omega \times y^{[1]}[k]$$

Figure 15.4c tracks the movement of the data values. At the beginning of the algorithm the vector elements are permuted. The element at index i of the input

```
Iterative_FFT (a, n):

  Parameter n          Number of elements in a
            a[0...(n-1)]  Coefficients
  Local     ω_d        Primitive dth root of unity
            ω          Evaluate polynomial at this point
            y          Result of transform

y ← Bit_Reverse_Permutation(a)
for j ← 1 to log n
  d ← 2^j
  ω_d ← e^(2πi/d)
  ω ← 1
  for k ← 0 to d/2 - 1
    for m ← k to n - 1 step d
      t ← ω × y[m + d/2]
      x ← y[k]
      y[k] ← x + t
      y[k + d/2] ← x - t
    endfor
  endfor
  ω ← ω × ω_d
endfor
return y
```

Figure 15.5 Iterative, sequential implementation of the fast Fourier transform algorithm (adapted from Cormen et al. [18]).

vector is moved to index $rev(i)$, where $rev(i)$ represents the bits of i in reverse order. Value 2, initially at index 01, is moved to index 10. Value 4, initially at index 10, is moved to index 01. Values 0 (at index 00) and 3 (at index 11) stay put. In the first stage the algorithm is finding the DFT of individual values, and it simply passes the values along. In each of the remaining stages the computation of a new value depends upon two values from the previous stage. The data flow arrows form **butterfly patterns.**

We can derive the iterative algorithm directly from Figure 15.4. After an initial permutation step, the algorithm will iterate $\log n$ times. Each iteration corresponds to a horizontal layer in Figure 15.4c. Within an iteration the algorithm updates values for each of the n indices. The algorithm, illustrated in Figure 15.5, has the same time complexity as the recursive algorithm: $\Theta(n \log n)$. The use of temporary variable t cuts the number of complex number multiplications nearly in half.

15.5 PARALLEL PROGRAM DESIGN

15.5.1 Partitioning and Communication

The efficient iterative algorithm is our starting point for designing a parallel FFT function suitable for implementation on a multicomputer. We'll use a domain

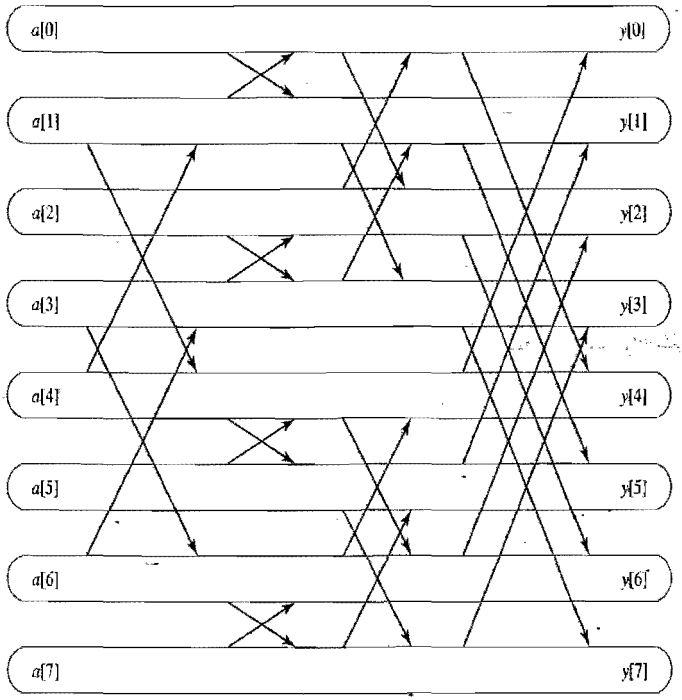


Figure 15.6 Task/channel graph for the FFT algorithm when $n = 8$. Tasks are represented by long, rounded rectangles so that the channels needed at each stage of the algorithm can be distinguished.

decomposition, associating a primitive task with each element of the input vector a and the corresponding element of the output vector y (Figure 15.6).

The first step of the algorithm is to perform the permutation of vector a. Each element $a[i]$ is copied to $y[j]$, where j is the index found by reversing the bits of i. For example, when $n = 8$, we have

$$\begin{aligned} 001 &\rightarrow 100 \\ 010 &\rightarrow 010 \\ 011 &\rightarrow 110 \\ &\ldots \\ 110 &\rightarrow 011 \\ 111 &\rightarrow 111 \end{aligned}$$

We draw channels for this initial communication.

The main loop of the function has $\log n$ iterations. During each iteration, each task computes its new value of $y[k]$ from the previous values of $y[k]$ and either $y[k+m/2]$ or $y[k-m/2]$. The task/channel graph illustrates the butterfly communication pattern.

15.5.2 Agglomeration and Mapping

Agglomerating primitive tasks associated with contiguous elements of the vector eliminates some of the communication steps. For example, if $n = 16$ and $p = 4$, process 0 has coefficients with subscripts 0, 1, 2, and 3; process 1 has coefficients with subscripts 4, 5, 6, and 7; and so on.

We can draw another task/channel diagram that indicates the communication pattern between the processes. In this new diagram (Figure 15.7), each agglomerated task (process) is represented by a gray rectangle. Every process controls two arrays of complex values. The first array, a, contains a contiguous group of input coefficients. The second array, y, holds intermediate values. At the end of the computation, array y contains a contiguous group of transformed values.

The parallel algorithm has three phases. In the first phase the processes permute the a's. This is an example of an all-to-all communication. In the second phase the processes perform the first $\log n - \log p$ iterations of the FFT by performing the required multiplications, additions, and subtractions on complex numbers. No message passing is required. In the third phase the processes perform the final $\log p$ iterations of the FFT by swapping y's and performing the requisite multiplications, additions, and subtractions. Think of the processes as being organized as a logical hypercube. During each of the final $\log p$ iterations, pairs of processes swap values across a different dimension of the hypercube.

15.5.3 Isoefficiency Analysis

Each process performs an equal share of the computations. Since the computational complexity of the sequential algorithm is $\Theta(n \log n)$, the computational complexity of the parallel algorithm is $\Theta(n \log n/p)$.

Each process controls at most $\lceil n/p \rceil$ elements of a. We assume the processes are organized as a logical hypercube. The all-to-all communication step is implemented as a series of swaps across each hypercube dimension; it has time complexity $\Theta[(n/p) \log p]$. There are $\log p$ iterations in which each process swaps about n/p values with a partner process along one of the hypercube dimensions. The total time complexity of these swaps is $\Theta[(n/p) \log p]$. With these assumptions, the overall communication complexity of the algorithm is $\Theta[(n/p) \log p]$.

Let's determine the isoefficiency of the parallel program. The sequential algorithm has time complexity $\Theta(n \log n)$. The parallel overhead is p times the communication complexity. Hence the isoefficiency function is

$$n \log n \geq Cn \log p \Rightarrow \log n \geq C \log p \Rightarrow n \geq p^C$$

The scalability of the FFT algorithm is similar to the scalability of the hyperquicksort and PSRS algorithms.

15.3 For each of the following vectors, show the result of applying the DFT to it.

a. $(7, 11)$

b. $(13, 17, 19, 23)$

c. $(2, 1, 3, 7, 5, 4, 0, 6)$

15.4 For each of the following vectors, show the result of applying the inverse DFT to it.

a. $(3, -2)$

b. $(10, -2 + 2i, -2, -2 - 2i)$

c. $(14, -3 - 4i, 1 - i, -1 + 3i, 0, -1 - 3i, 1 + i, -3 + 4i)$

15.5 Implement a parallel FFT program based on the design developed in this chapter. Benchmark your program for various values of n and p.

15.6 Implement a serial program implementing the inverse FFT algorithm.

15.7 Implement a parallel program implementing the inverse FFT algorithm. Benchmark your program for various problem sizes on various numbers of processors.

15.8 Excluding the initial all-to-all communication, the body of the fast Fourier transform algorithm exhibits a butterfly communication pattern. Name a parallel algorithm described in an earlier chapter that also has a butterfly communication pattern.

15.9 The scalability of the FFT algorithm is similar to the scalability of the hyperquicksort algorithm. Explain the similarities between the two algorithms.

CHAPTER

16

Combinatorial Search

Attempt the end, and never stand to doubt;
Nothing's so hard but search will find it out.
Robert Herrick, "Seek and Find," ***Hesperides***

16.1 INTRODUCTION

Combinatorial algorithms perform computations on discrete, finite mathematical structures [97]. **Combinatorial search** is the process of finding "one or more optimal or suboptimal solutions in a defined problem space" [109] and has been used for such diverse problems as:

- laying out circuits in VLSI to minimize the area dedicated to wires
- planning the motion of robot arms to minimize total distance traveled
- assigning crews to airline flights
- proving theorems
- playing games

There are two kinds of combinatorial search problems. An algorithm to solve a **decision problem** attempts to find a solution that satisfies all the constraints. The answer to a decision problem is either yes, meaning a solution exists, or no, meaning a solution does not exist. Here is an example of a decision problem: "Is there a way to route the robot arm so that it visits every drill site and moves no more than 15 meters?" An algorithm that solves an **optimization problem** must find a solution that minimizes (or maximizes) the value of an objective function. Here is an example of an optimization problem: "Find the shortest route for the robot arm that visits every drill site."

This chapter discusses four kinds of combinatorial search algorithms used to solve decision and optimization problems. These algorithms are divide and

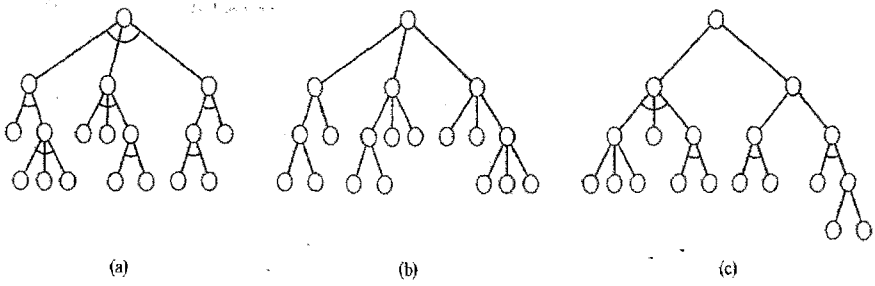


Figure 16.1 A search problem can be represented by a tree. (a) An AND tree. (b) An OR tree. (c) An AND/OR tree.

conquer, backtrack, branch and bound, and alpha-beta search. Different algorithms explore different kinds of search trees. In all cases the root of the search tree represents the initial problem to be solved, but the nonterminal nodes vary according to the kind of search tree. An AND node represents a problem or subproblem that is solved only when all its children have been solved; an OR node represents a problem or subproblem that is solved when any of its children has been solved. Every nonterminal node in an **AND tree** is an AND node (Figure 16.1a). The search tree corresponding to a divide-and-conquer algorithm is an AND tree, since the solution to a problem is found by combining the solutions to all its subproblems. Every nonterminal node in an **OR tree** is an OR node (Figure 16.1b). Backtrack search and branch-and-bound algorithms explore OR trees. An **AND/OR tree** is characterized by the presence of both AND nonterminal nodes and OR nonterminal nodes (Figure 16.1c). Game trees are examples of AND/OR trees.

16.2 DIVIDE AND CONQUER

Divide and conquer is a problem-solving methodology that involves partitioning a problem into subproblems, solving the subproblems, and then combining those solutions into a solution for the original problem. The methodology is recursive: that is, the subproblems themselves may be solved by the divide-and-conquer technique. The quicksort algorithm of Chapter 14 is an example of the divide-and-conquer technique.

The divide-and-conquer problem solution can be represented by an AND tree, since the solution to any problem represented by an interior node requires the solution of all its subproblems, represented by the children of that node. In other words, every node in the tree must be examined.

Divide-and-conquer algorithms are more easily implemented on centralized multiprocessors than on multicomputers. In a centralized multiprocessor the list of unsolved subproblems can be kept in a single stack manipulated by all the processors. Processors needing work can access the stack to retrieve an unsolved subproblem. Processors with extra subproblems can put them back in the stack for

other processors to retrieve. The central stack is an effective workload-balancing mechanism, though it eventually becomes a bottleneck as the number of processors increases.

In a multicomputer, the lack of a shared memory means subproblems must be distributed among the memories of the individual processors. Two fundamentally different designs emerge. In the first design the original problem and the final solution are stored in the memory of a single process. The parallel search of an AND tree can be divided into three phases. In the first phase, problems are divided and propagated throughout the parallel computer. For most of the first phase there are fewer tasks than processors, and processors are idle until they are given a problem to divide and propagate. In the second phase all the processors stay busy computing. In the third phase there are again fewer tasks than processors, and some processors combine results while other processors are idle. Hence the maximum speedup achievable is limited by the propagation and combining overhead.

In the second multicomputer design, both the original problem and the final solution are distributed among the memories of the processors. This design eliminates the starting up and winding down phases in which only some of the processors are active. It also allows the problem size to increase with the number of processors. We used this approach when developing parallel quicksort algorithms in Chapter 14. As we saw in our discussion of parallel sorting algorithms, it can be difficult to balance the workloads of the processors when subproblems are distributed among their memories.

16.3 BACKTRACK SEARCH

Backtrack is a method for solving combinatorial optimization problems that relies upon depth-first search to consider alternatives. Given the original problem (the root of the state space tree), backtrack generates its children and chooses one of them as the place to continue the search. It recursively applies the same methodology at the selected node. If the search reaches a node that cannot be expanded (i.e., a "dead end"), or if all of its children's subtrees have already been explored, then control backtracks to the previous node.

16.3.1 Example

Consider the problem of generating a crossword puzzle. Given a blank crossword puzzle (Figure 16.2a) and a dictionary of words and phrases, our goal is to assign letters to blank spaces so that every horizontal row and vertical column of two or more letters contains a word or phrase from the dictionary (Figure 16.2b). This is an example of a decision problem: we are answering the question, "Is there a way to fill in this particular crossword puzzle pattern with words from this particular dictionary?"

Each number in the blank crossword puzzle corresponds to the beginning of a horizontal word, the beginning of a vertical word, or both. We'll use the phrase "incomplete word" to refer to a word that has not yet been completely determined; that is, a word with at least one unassigned character in it.

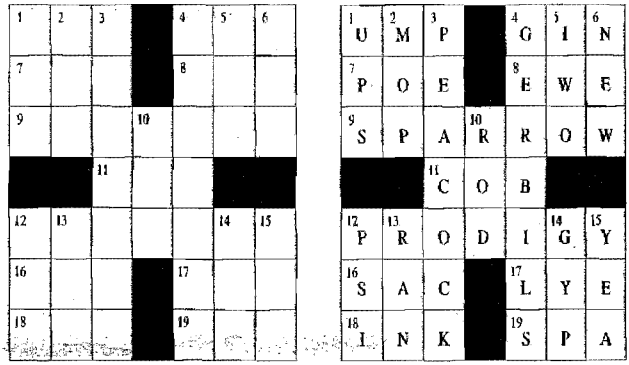


Figure 16.2 The crossword puzzle problem is to create a crossword puzzle solution from a dictionary of words and phrases and a blank puzzle template. (a) The blank puzzle template. (b) A crossword puzzle solution. This solution was generated by Crossword Weaver (www.CrosswordWeaver.com).

To fill in the puzzle, we use the following search strategy: We identify the longest incomplete word in the puzzle and look for a word or phrase in the dictionary of that length. If there is more than one word or phrase that fits, we choose one of them arbitrarily. At each subsequent step we locate the longest incomplete word that has at least one letter assigned and find a dictionary word of the correct length that matches the characters already assigned. Again, if there is more than word or phrase that would work, we select one arbitrarily. We continue in this fashion until no incomplete words remain.

With these criteria we can define the order in which we will try to fill in the incomplete words. Here is one word ordering that fulfills our criteria: 3 DOWN, 9 ACROSS, 4 DOWN, 12 ACROSS, 1 DOWN, 2 DOWN, 5 DOWN, 6 DOWN, 10 DOWN, 12 DOWN, 13 DOWN, 14 DOWN, 15 DOWN.

The different choices for the various word assignments can be represented by a **state space tree.** At the root of the tree is the empty crossword puzzle. The children of the root represent all the seven-letter words that can be used to fill the incomplete word 3 DOWN. Each node in the tree represents a possible dictionary element assignment to an incomplete word, given all the assignments that have been made so far. The tree has depth d if d assignments must be made to fill all the blank squares. Since any tree leaf at depth d represents a valid solution to the crossword puzzle problem, a state space tree is an example of an OR tree.

We can look for solutions to the crossword puzzle problem by performing a backtrack search of the state space tree. It has this name because if at any point in the search our choices lead us to a "dead end," we backtrack to the previous level and consider an alternate choice.

Let's see how this search would work for the blank crossword puzzle of Figure 16.2. Figure 16.3 accompanies the description that follows.

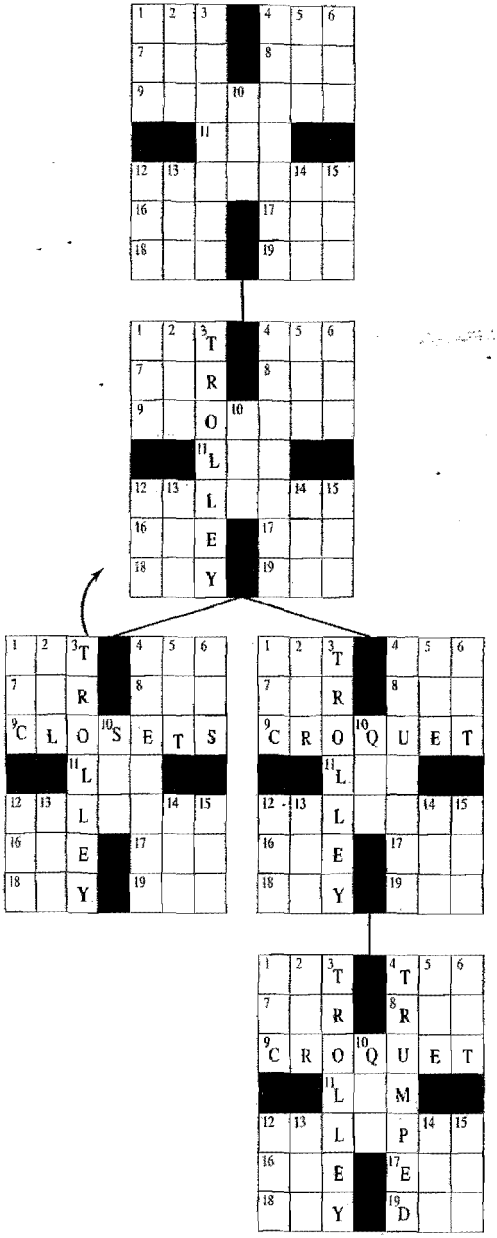


Figure 16.3 Start of a backtrack search of the state space tree for an instance of the crossword puzzle problem.

Our first step is to find a seven-letter word for 3 DOWN. Looking in the dictionary for a seven-letter word, we find TROLLEY.

Next we consider 9 ACROSS. Looking in the dictionary for a seven-letter word whose third letter is O, we find CLOSETS.

At the third level in the state space tree we consider 4 DOWN. We need a seven-letter word whose third letter is E. Suppose we can't find such a word in our dictionary. In this case, we must backtrack, and we look for another choice for 9 ACROSS. Another seven-letter word whose third letter is O is CROQUET.

Having filled in 9 ACROSS, we return to 4 DOWN, looking for a seven-letter word whose third letter is U. The dictionary word TRUMPED meets these criteria.

The search continues until no incomplete words remain or all possible alternatives have been exhausted for each word.

16.3.2 Time and Space Complexity

If the average branching factor in the state space tree is b, then searching a tree of depth k requires examining

$$1 + b + b^2 + \cdots + b^k = \frac{b^{k+1} - b}{b - 1} + 1 = \Theta(b^k)$$

nodes in the worst case. In other words, backtrack search of a state space tree takes exponential time in the worst case.

However, the amount of memory required by backtrack is linear in the depth of the search, or $\Theta(k)$, since only the currently chosen alternative at each level of the state space tree needs to be maintained in memory. Hence the size of the problem that can be solved by backtrack search is limited by the speed of the computer, not its primary memory capacity.

16.4 PARALLEL BACKTRACK SEARCH

Since we are dealing with an algorithm requiring exponential time in the worst case, there ought to be ample opportunities for parallelism. How can we perform backtrack search in parallel?

An obvious strategy is to divide the search of subtrees among the processes. See Figure 16.4. Suppose the tree has branching factor b and the number of processes $p = b^k$. Each process searches the state space tree to level k, and then explores only one of the subtrees rooted by a node at level k. If the depth of the search d is greater than $2k$, the time required for each process to traverse the first k levels of the state space tree is relatively small, and speedup can be high.

If there is no k such that $p = b^k$, the sequential search can go to level m in the state space tree, and each process can explore its share of the subtrees rooted by nodes at level m.

For example, suppose the branching factor of a state space tree is 3, and we are searching 10 levels deep. Suppose further we want to perform the search with

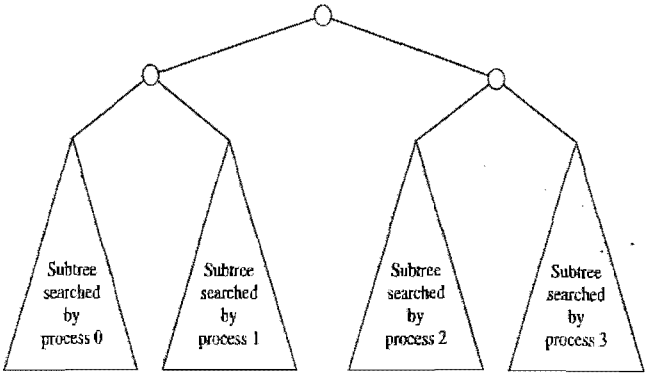


Figure 16.4 If $p = b^k$, then every process can search the state space tree to level k and then explore only one of the subtrees rooted by a node at level k. This is a good strategy if all the subtrees have the same number of nodes; it is a poor strategy if there are significant differences in the sizes of the subtrees.

five processes. If the parallel search starts at level 0 (the root) of the state space tree, there is only one node to search, and only one process has work to do. The resulting speedup of the search is 1.

If the parallel search starts at level 1 of the state space tree, there are three nodes to search, and three processes can be occupied. The time needed to expand the root node is negligible, and the resulting speedup is very close to 3.

If the parallel search starts at level 2 of the tree, there are nine nodes to search. Four processes examine two subtrees each, and one process examines the remaining subtree. The resulting speedup is very nearly $9/2 = 4.5$.

There is no value of k such that 3^k is a multiple of 5. However, as we go deeper into the tree, we generate a greater number of nodes, which means we can divide them up more evenly among the processes, improving speedup. On the other hand, going deeper into the tree means each process is spending more time generating the top levels. This is a redundant computation that increases the sequential portion of the overall computation. Figure 16.5 plots best-case speedup against the depth in the state space tree at which the parallel search begins. As you can see, the best-case speedup stays reasonably high throughout a broad range of depths.

Unfortunately, in most cases the state space tree is not balanced. Some subtrees have many more nodes than others. For example, in the case of the crossword puzzle problem, you can imagine that some early word choices would lead to dead ends and backtracking much sooner than other choices. Hence we need an algorithm that works reasonably well even when the tree is imbalanced.

One approach is to make the sequential search go deep enough in the state space tree that each parallel process is responsible for examining a large number of subtrees. This strategy is based upon a probabilistic argument: If each process handles a large number of subtrees, then the differences in the total time spent per

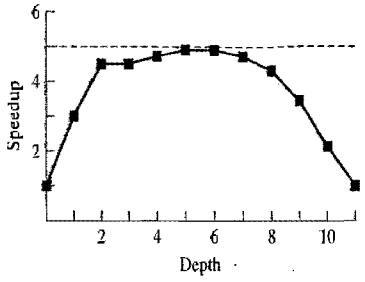


Figure 16.5 Maximum speedup achievable by five processes performing backtrack search of a state space tree with branching factor 3 and depth 10, as a function of the tree level at which the nodes are divided among the processes.

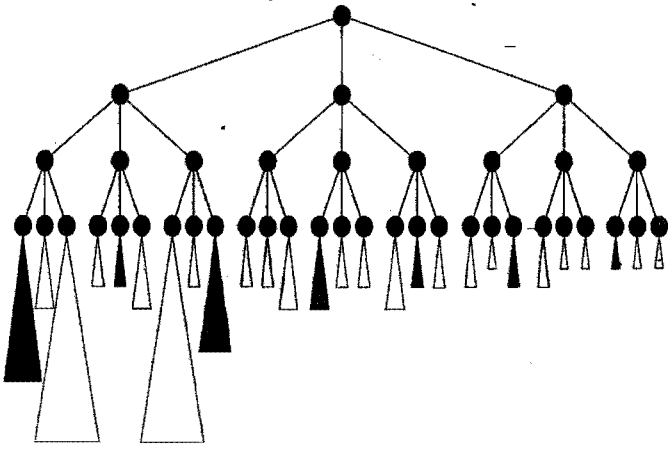


Figure 16.6 Parallel search of an unbalanced state space tree. Allocating a large number of subtrees to each process increases the probability that each process will search about the same number of nodes. In this example subtrees are allocated in an interleaved fashion to four processes. The nodes and subtrees explored by process 0 are black.

process are more likely to be smaller than if each process handles only a small number of subtrees.

Figure 16.6 illustrates how this strategy could be used to divide the search of an unbalanced state space tree among four processes. Process p_i searches the state space tree to level 3. It numbers the nodes at level 3 and continues the search only from those nodes whose numbers are equal to p_i modulo 4. The figure highlights in black the nodes and subtrees explored by process 0.

```
Global Variables:
  cutoff_count — Count of nodes at cutoff depth
  cutoff_depth — Depth at which subtrees are divided among processes
  depth — Depth to which state space tree is searched
  moves — Records position in search tree (i.e., moves made so far)
  id — Process rank
  p — Number of processes

Parallel_Backtrack (board, level):
  if level = depth then
    if board represents a solution to the problem then
      Print_Solution (moves)
    endif
  else
    if level = cutoff_depth then
      cutoff_count ← cutoff_count +1
      if cutoff_count mod p ≠ id then
        return
      endif
    endif
    possible_moves ← Count_Moves(board)
    for i ← 1 to possible_moves do
      Make_Move (board, i)
      moves[level] ← i
      Parallel_Backtrack (board, level+1)
      Unmake_Move (board, i)
    endfor
  endif
  return
```

Figure 16.7 Parallel backtrack search algorithm that divides subtrees among processes. This algorithm prints every solution.

Figure 16.7 gives pseudocode for a parallel backtrack search algorithm based on this approach. Every MPI process initializes variable *board* to represent the unsolved problem. It sets *level* to 0, because the search begins at the root of the state space tree, which is at level 0. It also assigns 0 to *cutoff_count*, the count of nodes the process has encountered at level *cutoff_depth* in the state space tree. Every process then calls function **Parallel_Backtrack** with actual parameters *board* and *level*. Processes search the entire tree to level *cutoff_depth*. Each process performs a backtrack search on its portion of the subtrees rooted at level *cutoff_depth*.

16.5 DISTRIBUTED TERMINATION DETECTION

Note that each process executing the parallel backtrack algorithm of Figure 16.7 only terminates after it has searched its portion of the entire state space tree to the specified depth. In other words, this algorithm finds every solution. When

using backtrack to solve an optimization problem, the processes must find every solution and then select the optimum solution.

However, at other times we only want a single solution. In these circumstances we would like all of the processes to halt as quickly as possible after one process has found a solution. How do the processes know when to stop searching? If we want processes to halt before they have completely searched their portions of the state space tree, a process that finds a solution must send a message—either directly or indirectly—to the other processes, and all processes must periodically check for messages. One way to add a periodic message check is to have each process look for a message every time the search reaches a particular level, such as *cutoff_depth*. Function `MPI_Iprobe` is a good choice to implement this message check, because it allows a process to determine, without blocking, if a message has arrived. It is easy enough to add this check to function **Parallel_Backtrack**.

A simple (but incorrect) approach to terminating the program is to have a process that finds a solution send a message to all of the other processes. A process halts after any one of the following events has occurred:

- It has found a solution and has sent a message to all of the other processes.
- It has received a message from another process.
- It has completely searched its portion of the state space tree.

Unfortunately, if a process calls `MPI_Finalize` before another, active process tries to send it a message, we get a run-time error. The approach we have suggested is subject to this error. How could this happen?

Here is one scenario that could lead to this particular run-time error. Suppose process A finds a solution, sends messages to the other processes, and calls `MPI_Finalize`. Meanwhile, process B finds another solution and sends messages to the other processes before it receives the message from process A. If process B tries to send a message to process A after process A has called `MPI_Finalize`, we will get a run-time error.

We must ensure that all processes are inactive and no messages are en route before we allow the processes to call `MPI_Finalize`. This is called the **distributed termination detection problem.** About 20 years ago Dijkstra, Seijen, and Gasteren invented an algorithm to solve this problem [21].

Figure 16.8 illustrates their algorithm. The processes are organized into a logical ring (Figure 16.8a). One process (in this example process 0) probes the state of the system by passing a token to its successor in the ring. When the token returns to process 0, it will be able to determine if it is safe for all the processes to terminate.

Each process has a color and a message count. When a process begins execution, it is white and its message count is zero. A process turns black when it sends or receives a message. When a process sends a message it increments its message count, and when a process receives a message it decrements its message count.

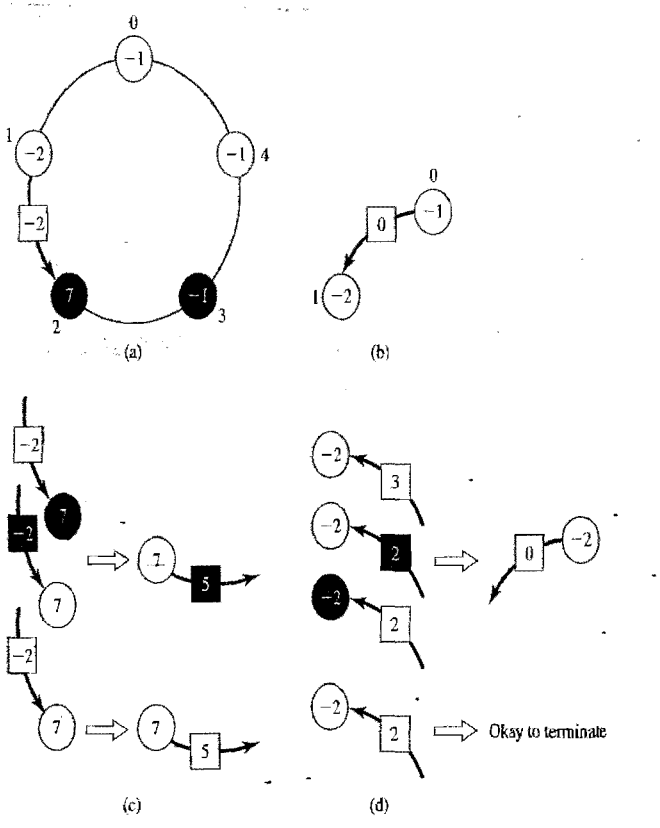


Figure 16.8 Dijkstra et al.'s algorithm to solve the distributed termination problem. (a) A token (square) is passed around a logical ring of processes (circles). (b) Process 0 initiates a probe. (c) An intermediate process modifies the token and passes it along. (d) The token returns to process 0.

The idea behind Dijkstra et al.'s algorithm is that if all processes are white and the sum of all message counts is zero, then we know there are no messages pending in the system, and we can terminate the processes.

The token being passed also has a color and a count. When process 0 initiates the probe, the token is white, and its count is 0 (Figure 16.8b).

Now let's look at what happens when an intermediate process handles the token (Figure 16.8c). When a process receives the token, it adds its message count to the count of the token. If the process is actively processing, it holds the token until it is inactive. At this point if the process is black, it changes the token to black. Otherwise, it does not change the color of the token. The process changes its own color to white and sends the updated token to its successor process.

Eventually the token returns to process 0 (Figure 16.8d). If the token is white, the process is white, and the sum of the token's count and process 0's message count is 0, then the system is quiescent, and it is safe to terminate the processes. (This can be done by having process 0 send a message to the other processes, telling them to exit.) Otherwise, process 0 must probe the ring of processes again.

For our parallel backtrack algorithm, we can implement distributed termination detection in the following way: All processes begin searching with their message counts set to zero. When a process finds a solution, it sends a "solution found" message to process 0 and sets its message count to 1. When process 0 receives a "solution found" message, it decrements its message count. After finding a solution or receiving a "solution found" message from another process, process 0 initiates a distributed termination detection probe by initializing the token and passing it to its successor process in the ring.

If a process is actively searching, it stops searching as soon as it receives a token, because receipt of a token means another process has found a solution. Hence we do not have to worry about a process hanging on to a token until it is inactive. Intermediate processes simply change the color of the token, if appropriate, modify the count associated with the token, and pass it on. Note that processes do not decrement or increment their message counts when receiving or sending a token.

When process 0 receives a token and determines the system is quiescent, it sends a "termination" message to the other processes and then calls `MPI_Finalize` and `exit`. The other processes can call `MPI_Finalize` and `exit` as soon as they receive a "termination" message.

16.6 BRANCH AND BOUND

The **branch-and-bound method** is a variant of backtrack that takes advantage of information about the optimality of partial solutions to avoid considering solutions that cannot be optimal.

16.6.1 Example

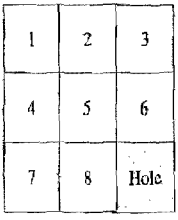


Figure 16.9 Goal state of the 8-puzzle, a simplified version of the 15-puzzle invented by Sam Loyd in 1878.

As an example of the branch-and-bound technique, consider the 8-puzzle (Figure 16.9), a simplified version of the well-known 15-puzzle invented by Sam Loyd in 1878. The 8-puzzle consists of eight tiles, numbered 1 through 8, arranged on a 3×3 board. Eight locations contain exactly one tile; the ninth location is empty. The object of the puzzle is to repeatedly fill the hole with a tile adjacent to it in the horizontal or vertical direction until the tiles are in row-major order. Unlike the kid at summer camp who is happy to slide tiles about until they are correctly ordered, our goal is to solve the puzzle in the least number of moves—an optimization problem.

We can use a state space tree to represent the board positions that can be reached from the initial position (Figure 16.10). One way to solve the puzzle is to pursue a breadth-first search of this state space tree until the sorted state is discovered. However, the goal is to examine as few alternative moves as possible.

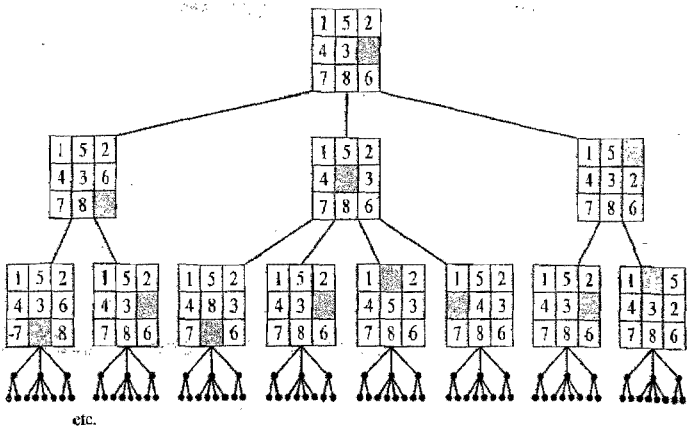


Figure 16.10 A portion of the state space tree corresponding to the search for a solution to a particular arrangement of the 8-puzzle.

We can achieve that goal and examine far fewer tree nodes if we associate with each state an estimate of the minimum number of tile moves needed to solve the puzzle, given the moves made so far.

One such function adds the number of tile moves made so far to the Manhattan distance between each out-of-place tile and its correct location (Figure 16.11). Given such a function, we can concentrate our search on the portions of the state space tree that contain the most promising moves. We always continue our search from the node having the smallest function value. If two or more nodes have the same value, we examine the node farthest from the root of the state space tree. If there are two or more nodes the same distance from the root with the same function value, we just pick one arbitrarily.

The branch-and-bound search of an example 8-puzzle appears in Figure 16.12. Let's look at how we assign to each node in the state space tree a lower bound on the cost of a solution going through that node. The first node we'll examine is the root of the tree. Looking at the state of the puzzle at the root, we see that tiles 2, 3, 5, and 6 are out of place by 1, 2, 1, and 1 positions, respectively (calculating the Manhattan distances). The sum $1 + 2 + 1 + 1 = 5$. Since 0 moves have been made so far, a lower bound on the cost of any solution is five moves. In other words, there is no way to solve the puzzle in fewer than five moves. (Since 5 is a lower bound, not an exact bound, it may take more than five moves to solve the puzzle.)

Now let's consider the left child of the root node. Tiles 2, 3, and 5 are out of place by one, two, and one positions, respectively. The sum $1 + 2 + 1 = 4$. Since one move has been made so far (the node is one level deep in the state space tree), a lower bound on the cost of any solution using this first move is 5.

Finally, let's consider the left child of the left child of the root node. Tiles 5, 2, 3, and 8 are out of place by a total of five positions. Since two moves have been

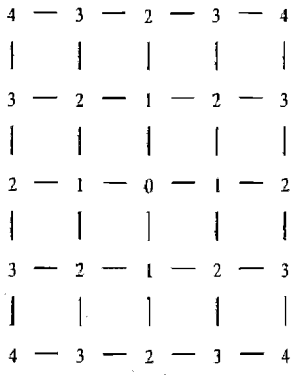


Figure 16.11 The Manhattan distance between a pair of points is the shortest path between those points when all movement must be in either the horizontal or vertical direction. This figure illustrates the Manhattan distance from the central intersection. (Imagine traveling from one intersection to another along a rectangular grid of streets.) Formally, the distance between points with coordinates (x_1, y_1) and (x_2, y_2) is $|x_1 - x_2| + |y_1 - y_2|$.

made so far, 7 is a lower bound on the cost of any solution using the two moves leading to this puzzle position.

The best-first search focuses on nodes with the smallest lower bounds. The search finds a solution to this particular puzzle in five moves. At this point there is no need to look for a better solution. We know from the lower bounds that all other solutions require at least seven moves. Note that the branch-and-bound search finds a solution with far fewer node examinations than would have been required if we had used a breadth-first search.

16.6.2 Sequential Algorithm

Now that we have seen a concrete example, let's develop a more general formulation of the branch-and-bound technique. Given an initial problem and some objective function f to be minimized, a branch-and-bound algorithm decomposes the problem into a set of two or more subproblems of smaller size. Every subproblem is characterized by the inclusion of one or more constraints. We repeat the decomposition process until each unexamined subproblem is decomposed, solved, or shown not be leading to an optimal solution to the original problem.

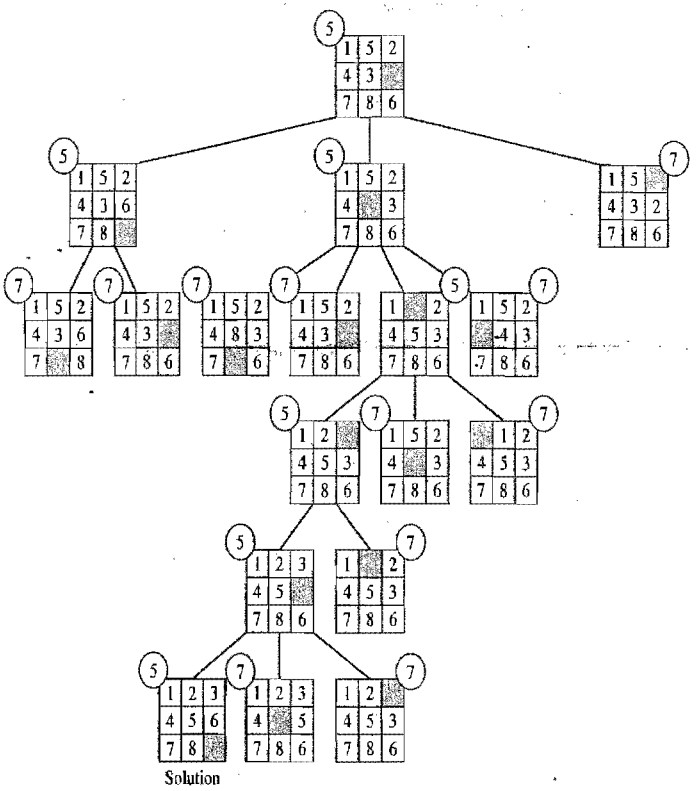


Figure 16.12 The best-first branch-and-bound search for a solution to an arrangement of the 8-puzzle. The nodes actually searched form a highly unbalanced tree.

In the 8-puzzle example, the problem is to put the pieces in order. The objective function f is the number of moves needed to order the pieces. If the pieces are in row-major order, the problem is solved. Otherwise, branch-and-bound decomposes the problem by generating a number of subproblems, one per legal move. Moving a tile represents the addition of a constraint.

As we have seen in the case of the 8-puzzle, we can represent the decomposition process applied to the original problem as a **state space tree.** The nodes of this tree correspond to the decomposed problems, and the arcs of the tree correspond to the decomposition process. The original problem is the root of the tree. The leaves of the tree are those partial problems that are solved or discarded without further decomposition.

Recall that the goal of the branch-and-bound technique is to solve the problem by examining a small number of elements in this tree. Assume that a minimum cost solution f^* is desired. We calculate a lower bounding function g for each decomposed subproblem as we create it. This lower bound represents the smallest

16.7.1 Storing and Sharing Unexamined Subproblems

The sequential algorithm keeps all unexamined subproblems in a priority queue. Maintaining a single priority queue on a distributed memory computer is impractical; the communication time required to send another processor an unexamined state space tree node and receive in return either a solution or the children of the node may well be greater than the time needed to do the computation locally. Giving a single processor responsibility for performing all priority queue manipulations also creates a performance bottleneck, limiting the maximum number of processors that can be applied to solving the problem. Finally, maintaining a single priority queue on one processor does not allow us to scale the problem size as the number of processors increases.

For these reasons, we make the design decision that each process must maintain its own priority queue of unexamined subproblems. In each of its iterations every process with a nonempty priority queue removes the unexamined subproblem with the smallest lower bound. If the subproblem is not a solution node, it divides it into b subproblems. (Note: Although each process iterates through a sequence of operations, there is no synchronization among processes.) If a process divides a problem into b subproblems, it puts the new subproblems into its priority queue.

Occasionally a process sends an unexamined subproblem to another process (Figure 16.15a). At the beginning of the program's execution, process 0 contains the original problem in its priority queue. The priority queues of the other processes are empty, and they have nothing to do. After process 0 distributes an unexamined subproblem, two processes may be active. After another distribution step, four processes may have subproblems to examine. If the distribution of unexamined subproblems is organized properly, $\lceil \log p \rceil$ distribution steps are sufficient to give all processes an unexamined subproblem.

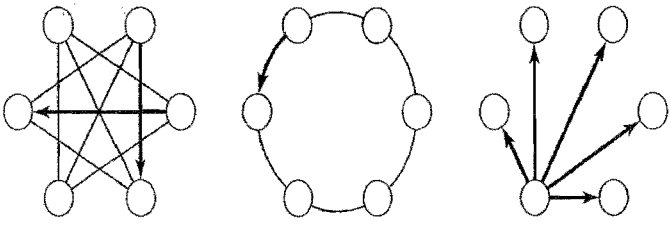


Figure 16.15 The parallel branch-and-bound algorithm uses three types of messages. (a) Processes send `UnexaminedSubproblem` messages to other processes. Each message contains an unexamined subproblem that the sending process deleted from its priority queue. The receiving process inserts the unexamined subproblem into its priority queue. (b) Processes form a logical ring to pass the token used for distributed termination detection. (c) Process 0 sends a termination message to all other processes when it is safe for them to exit.

16.7.2 Efficiency

In order for a solution to be found and guaranteed optimal, two conditions must be met. First, at least one of the solution nodes (and hence all its ancestors in the state space tree) must be examined. Second, processes must examine all nodes in the state space tree whose lower bounds are less than the cost of the optimal solution. The execution time of the algorithm is determined by whichever event occurs last. The event occurring last is determined by the number of processes and the shape of the state space tree.

The sequential best-first branch-and-bound algorithm, relying upon a single priority queue, examines the minimum number of nodes possible, given a particular bounding function g. It always examines the node with the smallest lower bound. Hence it can cease execution as soon as it encounters a solution node, because by definition no nodes with smaller lower bounds exist.

In contrast, the parallel best-first branch-and-bound algorithm may examine unnecessary nodes. That is because each process is examining the node that is only locally best—the node with the smallest lower bound in its local priority queue. While one process is guaranteed to be examining a node that is globally best, the rest of the processes may not. If a process examines a node whose lower bound is greater than the cost of the best solution, then examining that node is wasted work, and the overall efficiency of the parallel computation drops. The reason processes send out unexplored subproblems throughout the execution of the parallel algorithm is because it promotes the distribution of subproblems with good lower bounds among all the processes, reducing the amount of wasted work. On the other hand, passing around unexplored subproblems increases the communication overhead of the parallel algorithm.

16.7.3 Halting Conditions

Distributed termination detection for branch-and-bound algorithms is more complicated than for backtrack search. In the case of backtrack search we were simply looking for any solution. Now we are looking for an optimum solution. The first solution found by a process may not be the optimum solution. Hence we can only terminate when we have both (1) found a solution and (2) verified that no better solutions exist. Assume we are solving a minimization problem; that is, trying to find a solution of minimum cost. The two conditions are met when the cost of the best solution found so far is less than or equal to the lower bound on the cost of any solution from an unexamined subproblem.

We can solve this problem by modifying Dijkstra et al.'s distributed termination detection algorithm. A process turns black if it receives a message or manipulates an unexamined subproblem with a lower bound less than the cost of the best solution found so far. How does a process know the cost of the best solution found so far? We add additional information to the termination token passed around the logical ring of processes (Figure 16.15b).

Recall that in the original algorithm the termination token had a count and a color. Now we add two additional fields: the cost of the best solution found so

far and the solution itself (i.e., the moves made to reach the solution). When a process receives the token, it updates the color and count fields. It also checks to see if the best solution it has found has a lower cost than the solution carried by the token. If so, it updates the token so that it now carries a better solution with its cost. Finally, the process compares the cost of the best solution found so far with the lower bound associated with the unexamined subproblem at the head of the priority queue. If this unexamined subproblem's lower bound is greater than or equal to the cost of the best solution found so far, there is no point in exploring it or any of the other nodes in the priority queue, since they cannot lead to a solution better than the best solution found so far. In this case, the process should re-initialize (i.e., empty) its priority queue.

With these modifications, process 0 still uses the same check to identify when the parallel algorithm has terminated. If it is white when it receives a white token, and if the sum of the token's count and process 0's message count is zero, then all work on the computation has ceased. Process 0 sends a termination message to all other processes (Figure 16.15c), and all processes can call functions `MPI_Finalize` and `exit`.

In effect, the token being passed around the ring serves two purposes. Intially its purpose is to keep the processes abreast of the value of the best solution found so far. Eventually, every process empties its priority queue when it discovers it cannot possibly find a solution better than the best solution already discovered. At this point the token's purpose is distributed termination detection: ensuring that all processes are inactive and all messages containing unexamined subproblems have been delivered.

The pseudocode for our parallel best-first branch-and-bound algorithm appears in Figure 16.16.

16.8 SEARCHING GAME TREES

The most successful computer programs to play two-person zero-sum games of perfect information, such as chess, checkers, and go, have been based on exhaustive search algorithms. These algorithms consider series of possible moves and countermoves, evaluate the desirability of the resulting board positions, then work their way back up the tree of moves to determine the best initial move.

16.8.1 Minimax Algorithm

Given a trivial game, the **minimax** algorithm can be used to determine the best strategy. Figure 16.17a represents the **game tree** of a hypothetical game, with rules left unstated, played for money. Dotted edges represent moves made by the first player; solid lines represent moves made by the second player. The root of the tree is the initial condition of the game. The leaves of this game tree represent outcomes of the game. Interior nodes represent intermediate conditions. The outcomes are always put in terms of advantage to the first player. Thus positive numbers indicate the amount of money in dollars won by the first player, while negative numbers indicate the amount of money lost by the first player. The algorithm assumes that

```
Constants:
  Comm_Interval — Time between communication steps
  Termination — Tags termination messages
  Token — Tags token message
  Unexamined_Subproblem — Tags message containing unexamined subproblem

Functions:
  Current_Time() — Wall clock time
  Delete_Min() — Delete subproblem with least lower bound from priority queue
  First_Element() — Returns first element from priority queue without deleting it
  Initialize() — Set priority queue size to 0
  Insert() — Insert subproblem into priority queue
  Is_Empty() — Returns true if priority queue is empty
  Lower_Bound() — Returns lower bound associated with unexplored subproblem

Variables:
  color — Process color (for termination detection)
  global_c — Cost of globally best solution found so far
  id — Process rank
  initial — Initial problem
  last_comm — Time of last communication
  local_c — Cost of best solution found so far by this process
  local_s — Best solution found so far by this process
  msg_count — Messages sent minus messages received
  q — Priority queue
  token — Token passed around ring for termination detection
  u — State space tree node
  v — New node with additional constraint

Parallel Best-First Branch and Bound (minimization):
Initialize (q)
if id = 0 then
  Insert (q, initial)
  token.c ← ∞
  token.color ← WHITE
  token.count ← 0
  Send token to successor process
endif
local_c ← ∞
best_soln ← ∞
last_comm ← Current_Time()
msg_count ← 0
color ← WHITE
repeat
  if Is_Empty(q) or (Current_Time()−last_comm > Comm_Interval) then
    BandB_Communication()
    last_comm ← Current_Time()
  else if not Is_Empty(q) then
    u ← Delete_Min(q)
    if Lower_Bound(u) < best_c then
      color ← BLACK
      if u is a solution then
        if Lower_Bound(u) < global_c then
          local_s ← u
          local_c ← Lower_Bound(local_s)
        endif
```

Figure 16.16 Parallel branch-and-bound algorithm.

```
      else
        for i ← 1 to Possible_Constraints(u) do
          Add constraint i to u, creating v
          if Lower_Bound(v) < global_c then
            Insert(q, v)
          endif
        endfor
      endif
    endif
  endif
forever

BandBCommunication():
if there is a pending message with a Termination tag then Halt endif
if there is a pending message with a Token tag then
  Receive message containing token
  if local_c < token.c then
    token.c ← local_c
    token.s ← local_s
  endif
  if token.c ≤ Lower_Bound(First_Element(q)) then Initialize(q) endif
  global_c ← token.c
  if id = 0 then
    if (color = WHITE) and (token.color = WHITE) and
        (token.count + msg_count = 0) then
      Send messages with a Termination tag to all other processes
      Halt
    else
      token.color ← WHITE
      token.count ← 0
    endif
  else
    if color = BLACK then token.color ← BLACK
    token.count ← token.count + msg_count
  endif
  Send token to successor
  color ← WHITE
endif
while there are pending messages with tag Unexamined_Subproblem do
  Receive message with unexamined subproblem u
  msg_count ← msg_count − 1
  color ← BLACK
  if Lower_Bound(u) < global_c then Insert (q, u)
endwhile
if there is more than one unexamined subproblem in q then
  Send unexamined subproblem to another process
  msg_count ← msg_count +1
  color ← BLACK
endif
return
```

Figure 16.16 (contd.) Parallel branch-and-bound algorithm.

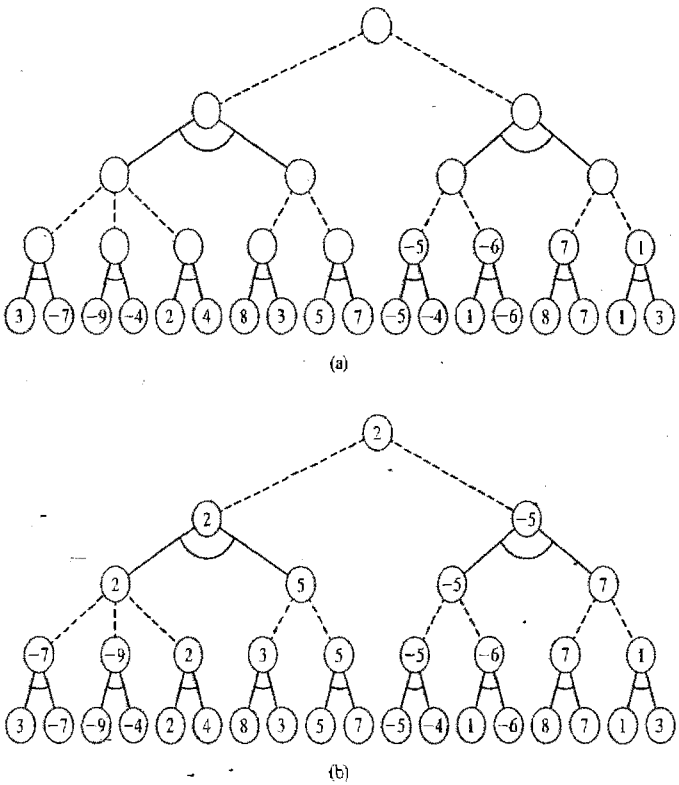


Figure 16.17 (a) A game tree. Dashed edges represent the moves available to the first player; solid edges represent moves available to the second player. (b) The same tree with the values of the interior nodes filled in. By taking the first alternative, the first player is guaranteed a result of at least 2.

the second player tries to minimize the gain of the first player, while the first player tries to maximize his or her own gain, hence the name of the algorithm. Figure 16.17b is the same tree with the values of the interior nodes filled in. The value of this game to the first player is 2. If the first player plays the minimax strategy, he or she is guaranteed to win at least two dollars.

A game tree is an example of an AND/OR tree. The player moving first is the one evaluating the tree. The AND nodes represent positions where it is the second player's turn to move. In order to protect herself, the first player must consider every move an opponent might make. The OR nodes represent positions where it is the first player's turn to move. The first player does not need to consider every possible move, if she has already found a good one.

Nontrivial games such as chess have game trees that are far too complicated to be evaluated exactly. For example, de Groot has estimated that there may be 38^{84}

positions in a chess game tree [19]. Thus current chess-playing programs examine moves and countermoves only to a certain depth, then, at that point, estimate the value of the board position to the first player. Of course, evaluation functions are imperfect. If a perfect evaluation function existed, the need for searching would be eliminated.

As we have seen, all possible moves from a position to some predetermined look-ahead horizon can be represented by a game tree. We can find the minimax value of a game tree by applying the evaluation function to the leaves of the tree (the terminal nodes), then working backward up the tree. If it is the second player's move at a particular nonterminal node in the game tree, the value we assign is the minimum over all its children nodes. If it is the first player's move, we assign the value that is the maximum over all its children nodes. Given a game tree in which every position has b legal moves, it's easy to see that a minimax search of the game tree to depth d requires an examination of b^d leaves.

16.8.2 Alpha-Beta Pruning

As a rule, the deeper the search, the better the quality of play. That is why **alpha-beta pruning** is valuable. Alpha-beta pruning, a form of branch-and-bound algorithm, avoids searching subtrees whose evaluation cannot influence the outcome of the search, that is, cannot change the choice of best move. Hence it allows a deeper search in the same amount of time.

The alpha-beta algorithm, displayed in Figure 16.18, is called with four arguments: *pos,* the current condition of the game; α and β, the range of values over which the search is to be made; and *depth,* the depth of the search that is to be made. The function returns the minimax value of the position *pos*. The original game position is a MAX-NODE. Every child of a MAX-NODE is a MIN-NODE. Every child of a MIN-NODE is a MAX-NODE.

To illustrate the workings of the alpha-beta algorithm, consider the game tree in Figure 16.19. This tree represents the same game as that in Figure 16.17, except that nodes not examined by the alpha-beta algorithm are not included. When the algorithm begins execution, $\alpha = -\infty$ and $\beta = \infty$. The algorithm traverses the nodes of the game tree in preorder (i.e., depth first); the values of α and β converge as the search progresses.

The nodes drawn in heavy lines in Figure 16.19 represent places where **pruning** (elimination of the search of a subtree) occurs. To explore the conditions under which pruning happens, let's consider an arbitrary interior node in the search tree. When the search reaches this node, we know that some sequences of moves already considered leads to a value of at least α for the player moving first. We also know that correct play on the part of the opponent will ensure that the first player cannot get a value more than β. Hence α and β define a window for the search.

If the interior node *pos* is a MAX-NODE, then it is the first player's move. If *val,* the value of the game tree searched from *pos,* is greater than α, then α is changed to *val,* meaning a better line of play has been found for player 1.

```
Constant:
  max.c — Maximum possible number of moves

Alpha_Beta (pos, α, β, depth):

Parameters:
  pos — Position
  α — Lower cutoff value
  β — Upper cutoff value
  depth — Search depth

Variables:
  c[1...max.c] — Children of pos in game tree
  cutoff — Set to TRUE when okay to prune
  i — Iterates through legal moves
  val — Value returned from search
  width — Number of legal moves

begin
  if depth ≤ 0 then
    return (Evaluate(pos)) {Evaluate terminal node}
  endif
  width ← Generate_Moves(pos)
  if width = 0 then
    return (Evaluate(pos)) {No legal moves}
  endif
  cutoff ← FALSE
  i ← 1
  while (i ≤ width) and (cutoff = FALSE) do
    val ← Alpha_Beta(c[i], α, β, depth−1)
    if Max_Node(pos) and val > α then
      α ← val
    elseif Min_Node(pos) and val < β then
      β ← val
    endif
    if α > β then
      cutoff ← TRUE
    endif
    i ← i + 1
  endwhile
  if Max_Node(pos) then return α
  else return β
  endif
end
```

Figure 16.18 Sequential alpha-beta pruning algorithm.

Analogously, if the interior node *pos* is a MIN-NODE, then it is the second player's move. If *val*, the value of the game tree searched from node *pos* is less than β, then β is changed to *val;* a better line of play has been found for player 2.

However, if at any time the value of α exceeds the value of β, there is no need to search further, because it is in the best interests of one of the players to block the line of play leading to the position (node) being considered.

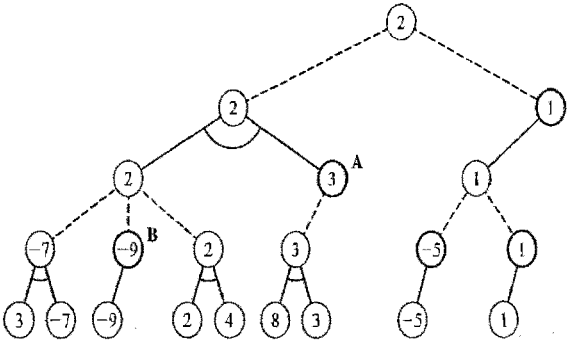


Figure 16.19 An illustration of alpha-beta pruning. The number inside each node is the value of the position. In the case of leaf nodes, an evaluation function computes the value of the position. In the case of interior nodes, the value is computed from the values of its children. Highlighted circles represent nodes at which pruning occurs. Note how many fewer nodes are examined than in minimax search (Figure 16.17).

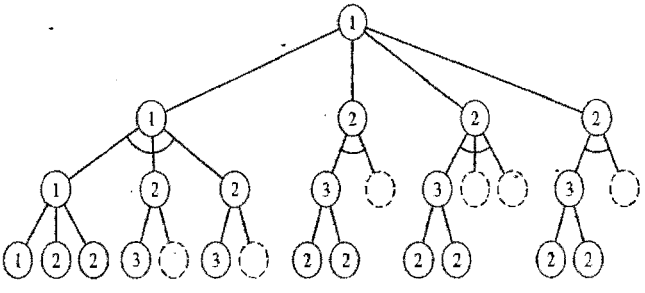


Figure 16.20 Alpha-beta pruning of a perfectly ordered game tree. The number inside each node indicates its classification as either type 1, type 2, or type 3. The root of the tree is a type 1 node. The first child of a type 1 node is a type 1 node. All other children of a type 1 node are type 2 nodes. The first child of a type 2 node is a type 3 node; all other children of a type 2 node may be pruned. All children of a type 3 node are type 2 nodes.

For example, consider the node labeled **A** in Figure 16.19. The value returned from the search of the first child of **A** is 3, which is greater than 2, the value of β. It is not in the second player's interest to allow play to reach this position, since there is another line of play guaranteeing a value no higher than 2. Hence there is no point in continuing the search from this game position.

To what extent can alpha-beta pruning reduce the number of leaf nodes that must be examined? The algorithm does the most pruning on a **perfectly ordered game tree,** that is, a game tree in which the best move from each position is always examined first (see Figure 16.20). Assuming a perfectly ordered game tree with

a search depth of d and uniform branching factor b, Slagle and Dixon [103] have shown that the number of leaf nodes examined by the alpha-beta algorithm is

$$Opt(b, d) = b^{\lceil d/2 \rceil} + b^{\lfloor d/2 \rfloor} - 1$$

In other words, in the best case it is possible for the alpha-beta algorithm to examine no more than approximately twice the square root of the number of nodes searched by the minimax algorithm.

The **effective branching factor** of an algorithm searching a game tree of depth d is the dth root of the number of leaf nodes evaluated by the algorithm. An alpha-beta search reduces the effective branching factor from b to $\sqrt{b}$ when searching a perfectly ordered game tree. In other words, in the best case alpha-beta pruning allows the game tree search to go twice as deep in the game tree as the minimax algorithm in the same amount of time. Experimental evidence indicates that sequential alpha-beta algorithms often search no more than 50 percent more nodes than would be searched if the tree were perfectly ordered. Hence in practice the alpha-beta search algorithm exhibits much higher performance than minimax.

16.8.3 Enhancements to Alpha-Beta Pruning

Two common enhancements to alpha-beta pruning are aspiration search and iterative deepening. **Aspiration search** makes an estimate of the value v of the board position at the root of the game tree, figures the probable error e of that estimate, then calls the alpha-beta algorithm with the initial window $(v-e, v+e)$. If the value of the game tree does indeed fall within this window of values, then the search will end sooner than if the algorithm has been called with the initial window $(-\infty, \infty)$. If the value of the game tree is less than $v-e$, the search will return the value $v-e$, and the algorithm must be called again with another window, such as $(-\infty, v-e)$. Similarly, if the value of the game tree is greater than $v+e$, the search returns the value $v+e$, and another search will have to be done with a modified initial window, such as $(v+e, \infty)$.

Another variant on the standard alpha-beta algorithm is called **iterative deepening.** Each level of a game tree is called a **ply** and corresponds to the moves of one of the players. Iterative deepening is the use of a $(d-1)$-ply search to prepare for a d-ply search. This technique has three advantages. First, it allows the time spent in a search to be controlled. The search can be continued deeper and deeper into the game tree until the allotted time has expired. Second, results of the $(d-1)$-ply search can be used to improve the ordering of the nodes during the d-play search, making the node ordering similar to the perfect ordering, and increasing the amount of pruning. Finally, the value returned from a $(d-1)$-ply search can be used as the center of the window for a d-ply aspiration search.

16.9 PARALLEL ALPHA-BETA SEARCH

Alpha-beta search has a number of opportunities for parallel execution. One approach is to parallelize move generation and position evaluation. The custom chess machine HITECH™, with 64 processors organized as an 8×8 array, is an

example of this approach. However, the speedup that can be achieved with this approach is limited by the parallelism inherent in these activities.

Further speedup improvements lie in parallelizing the search. This is the approach taken by the team that programmed IBM's Deep Blue, a 32-node RS/6000 multicomputer augmented with 192 VLSI chess processors. Capable of searching more than 100 million positions per second, Deep Blue defeated world chess champion Gary Kasparov in a six-game match in 1997 by the score of 3 1/2 to 2 1/2.

16.9.1 Parallel Aspiration Search

A straightforward parallelization of the alpha-beta algorithm is done by performing an aspiration search in parallel. If three processors are available, then each processor can be assigned one of the windows $(-\infty, v - e)$, $(v - e, v + e)$, and $(v + e, \infty)$. Ideally the processor searching $(v - e, v + e)$ will succeed, but all three processors will finish no later than a single processor searching the window $(-\infty, \infty)$. We can accommodate additional processors by creating narrower windows.

Experiments with parallel aspiration search for the game of chess has led to two conclusions. First, the maximum expected speedup is typically five or six, regardless of the number of available processors. This is because $Opt(b, d)$ is a lower bound on the cost of alpha-beta search, even when both α and β are initially set to the value eventually returned from the search. Second, parallel aspiration search can sometimes lead to superlinear speedup when two or three processors are being used.

16.9.2 Parallel Subtree Evaluation

Another approach is to allow processors to examine independent subtrees in parallel. When taking this approach, we must consider two overheads. **Search overhead** refers to the increase in the number of nodes that are examined owing to the introduction of parallelism. **Communication overhead** refers to the time spent coordinating the processes performing the search. Search overhead can be reduced at the expense of communication overhead by keeping every processor aware of the current search window (α, β). Communication overhead can be reduced at the expense of search overhead by allowing processors to work with outdated search windows.

For example, consider this simple method of performing alpha-beta search in parallel. Split the game tree at the root, and give every processor an equal share of the subtrees. Let every processor perform an alpha-beta search on its subtrees. Each processor begins with the search window $(-\infty, \infty)$, and no processor ever notifies other processors of the changes in its search window. Clearly this algorithm minimizes communication overhead. What is the speedup achievable by this method?

Slagle and Dixon showed that in a perfectly ordered uniform game tree of depth d and branching factor b, the number of node examinations performed by alpha-beta search is $Opt(b, d)$. We can use the same formula to determine that the number of node examinations in the first branch of a perfectly ordered game tree is $Opt(b, d-1)$.

What this means is that the examination of the first branch of a perfectly ordered game tree takes a disproportionate share of the computation time. For example, consider a 10-ply search of a perfectly ordered tree that has a branching factor of 38 (such as a chess game tree). The minimum number of node examinations is 158,470,335. The minimum number of node examinations in the first branch is 81,320,303. By Amdahl's Law it is clear that if only one processor is responsible for searching the first move's subtree, speedup will be less than 2.

In addition, because every processor's search must begin with $-\infty$ and ∞ as the values of α and β, respectively, the parallel algorithm will not prune as many subtrees as the sequential algorithm. A complete elimination of communication overhead creates significant search overhead.

Let's look at the other extreme. What must be done to eliminate search overhead completely? We will make the assumption that the game tree is perfectly ordered. Look at Figure 16.20. If we want to eliminate search overhead, we must ensure that the parallel algorithm prunes the same nodes as the sequential algorithm. First consider searching the subtree of a type 1 node. The first child is a type 1 node; the remaining children are type 2 nodes. Searching subtrees rooted by type 2 nodes requires up-to-date values of α and β in order to prune all but the first children of the type 2 nodes. To get up-to-date values, the search of the subtrees rooted by type 2 nodes cannot begin until the search of the subtree rooted by the type 1 node has finished, returning α and β. However, once the values of α and β are known, all type 2 nodes may be searched in parallel without processor interaction.

In practice, search trees are not perfectly ordered, but this study has demonstrated that a parallel alpha-beta algorithm can significantly reduce search overhead by delaying the search of some subtrees until more accurate bounds information is available. That is the basis for our next algorithm.

16.9.3 Distributed Tree Search

Ferguson and Korf [26] have developed a parallel tree searching algorithm called Distributed Tree Search (DTS), which, when evaluating game trees, has achieved good speedups. Although the DTS algorithm is suitable for solving a variety of tree search problems, we will describe its use as a tool to perform parallel alpha-beta search.

The DTS algorithm executes by assigning processes to nodes of the search tree. Each process controls one or more physical processors. When the algorithm begins execution, a single process, called the root process, is assigned to the root node of the search tree. It controls the entire set of physical processors performing the search.

When a process is assigned to a nonterminal node, it generates the children of that node by evaluating the legal moves. The process assigns processors to the children nodes based upon the processor allocation strategy. The **bound-and-branch** strategy corresponds closely to the algorithm described at the end of the previous subsection. When the search reaches a type 1 node, all processors are allocated to the leftmost child. After the search returns with cutoff bounds from the subtree rooted by the leftmost child, the processors are assigned to the remaining children in a breadth-first manner. When the search reaches a node having type 2 or 3, cutoff bounds already exist, and the processors are assigned to children nodes in a breadth-first fashion. At this point a new process is created for each child node that is allocated at least one processor. The parent process suspends operation until it receives a message from another process (either one of its children or its parent).

When a process is assigned to a terminal node, it returns the value of that node and its set of allocated processors to the parent, then terminates.

The first child process to complete the search of its subtree sends a message with its values of α and β to the parent. It returns a set of processors to the parent and terminates. The parent process wakes up when it receives the message from its child. It reallocates the freed processors to one or more of its active child processes. It may also send one or more of its child processes new values of α and β. The reallocation of processors from quicker processes to slower processes produces efficient load balancing. Notice that in this scheme a child process may be awakened by its parent, which is passing along additional processors. After reallocating processors, parent processes suspend operation until they receive another message. When all child processes have terminated, the parent process returns α, β, and the set of processors to its parent and terminates. When the root process terminates, the algorithm has been completed.

Three implementation details improve the performance of the DTS algorithm. First, every blocked process should share a physical processor with one of its child processes. In this way all processors stay busy. Second, when a blocked parent process is awakened, it should have a higher priority for execution than processes corresponding to nodes deeper in the search tree. Third, when the search reaches a point where there is only a single processor allocated to a node, the process controlling the processor should execute the standard sequential alpha-beta search algorithm.

Given a uniform game tree with branching factor b, if the alpha-beta algorithm searches the tree with effective branching factor b^x (where $0.5 \leq x \leq 1$), then DTS with p processors and breadth-first allocation will achieve a speedup of $O(p^x)$.

To test the DTS algorithm, Ferguson and Korf [26] have implemented the game of Othello. Their node-ordering function results in an effective branching factor of about $b^{0.66}$. The program implements parallel alpha-beta search using the DTS algorithm. Executing the program on 40 midgame positions on a first-generation multicomputer, they reported a speedup of about 12 on 32 processors.

16.10 SUMMARY

Combinatorial search is used to find solutions to a variety of decision and optimization problems on discrete, finite mathematical structures. One way to differentiate between combinatorial search problems is to categorize them by the kind of state space tree they traverse. Divide-and-conquer algorithms traverse AND trees; the solution to a problem or subproblem is found only when the solution to all its children is found. Backtrack and branch-and-bound algorithms traverse OR trees; the solution to a problem or subproblem can be found without exploring every subproblem. Two-person games can be represented by AND/OR trees combining both kinds of nodes.

One way to think of parallel divide-and-conquer is to imagine that a single process is responsible for the computation that divides a problem (or subproblem) into pieces and combines the solutions to the subproblems. The speedup that can be achieved this way is limited by the propagation and combining overhead. In contrast, if the original problem and the final solution are decomposed among processors, then the efficiency of a parallel divide-and-conquer algorithm can be much higher. However, balancing workloads among processors can be a significant challenge.

Backtrack is depth-first search methodology for exploring state space trees. It can be used to find a single solution to a problem or every possible solution. It does not take advantage of knowledge about the problem to avoid exploring subtrees that cannot possibly lead to a solution. It has the advantage of only requiring space linear in the depth of the search. Since state space trees are often unbalanced, the principal challenge in parallel backtrack is providing every process with the same amount of work. We discussed the strategy of assigning many subtrees to each process, which increases the probability that the total number of nodes searched by each process will be roughly equivalent.

Ensuring that parallel backtrack terminates without a run-time error requires that the processes perform distributed termination detection. Dijkstra et al.'s algorithm allows the processes to detect quiescence in $\Theta(p)$ time.

Sequential branch-and-bound algorithms find solutions to combinatorial optimization problems much faster than exhaustive search algorithms such as depth-first search or breadth-first search, because they can bypass the examination of subtrees that cannot possibly lead to a solution. However, the fact that the state space trees actually explored by branch-and-bound algorithms have irregular shapes makes it difficult to assign processors to subtrees so that their workloads are balanced. The fundamental problem faced by designers of parallel branch-and-bound algorithms is keeping the efficiency of the processors high by focusing the search on the nodes the sequential algorithm examines.

Alpha-beta pruning is the preferred method for evaluating game trees. In the best case it allows the computer to look ahead twice as many moves as it would have time to explore using the brute-force minimax algorithm. Its performance can be further improved through the use of aspiration search and iterative deepening. We examined several methods to parallelize alpha-beta search: parallel move

16.8 If a perfect evaluation function existed, the need for searching a game tree would be eliminated. Explain.

16.9 Use the minimax algorithm to evaluate the game tree of Figure 16.21.

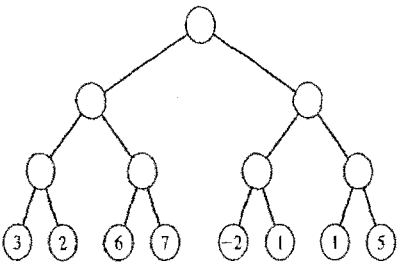


Figure 16.21 A game tree.

16.10 Explain why alpha-beta algorithm prunes the game tree of Figure 16.19 at the node labeled **B**.

16.11 Use the alpha-beta algorithm to evaluate the game tree of Figure 16.21.

16.12 Extend the perfectly ordered game tree of Figure 16.20 by one level to illustrate how nodes are pruned at level 4. Assume a branching factor of 2.

16.13 Explain why alpha-beta search is simply a special case of DTS.

16.14 How does improving the pruning effectiveness of the underlying alpha-beta algorithm affect the speedup achieved by the distributed tree search (DTS) algorithm on a particular application?

16.15 The ***N* queens problem** is to place N queens on an $N \times N$ chessboard so that no queen may attack another. Figure 16.22 illustrates a solution to the four queens' problem. Write a parallel program that counts the number of solutions to the N queens problem for a particular value of N input from the command line. Benchmark your program for various

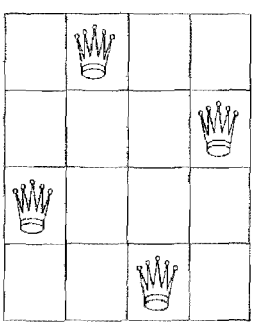


Figure 16.22 A solution to the four queens problem.

values of N and p. Plot the speedup achieved by your program as a function of N and p.

16.16 Write a parallel program that finds a single solution to the N queens problem for a particular value of N input from the command line. After printing the solution, the program should terminate. Benchmark your program for various values of N and p. Plot the speedup achieved by your program as a function of N and p.

16.17 Figure 16.23 illustrates a puzzle. The puzzle has 21 holes in it. Initially every hole is filled with a peg, except for the center hole (shown in black). Pegs are moved and removed by doing checkers-style hopping. You are allowed to move a peg in a straight line from its hole over an occupied hole to an empty hole on the other side and remove the peg that was just hopped. You may hop pegs in a horizontal, vertical, or diagonal direction.

The object of the puzzle is to remove pegs until only one peg remains, and that peg is in the center hole. A sequence of 19 moves (hops) is necessary to reduce the original 20 pegs to a single peg.

Write a parallel program to find a solution to the puzzle. Benchmark the execution time of your program for various values of p. Plot the speedup achieved by your program as a function of p.

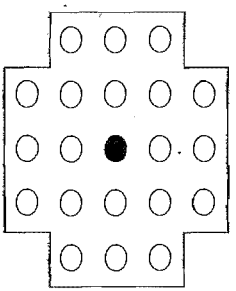


Figure 16.23 A peg puzzle.

16.18 The 15-puzzle, invented by Sam Loyd, is a larger version of the 8-puzzle presented in this chapter. Fifteen tiles, numbered 1 through 15, and a hole occupy a 4×4 grid. Write a parallel program that takes as input a scrambled version of the 15-puzzle and finds the shortest sequence of moves needed to put the tiles back in order. Benchmark your program on at least five puzzles, each of which requires at least six moves to solve. Plot the speedup achieved by your program on each puzzle, as a function of p.

16.19 Write a parallel program that plays the game of Othello (also called Reversi) against a human opponent.

CHAPTER

17

Shared-Memory Programming

Not what we give, but what we share—
For the gift without the giver is bare;
Who gives himself with his alms feeds three—
Himself, his hungering neighbor, and me.
James Russell Lowell, ***The Vision of Sir Launfal***

17.1 INTRODUCTION

In the 1980s commercial multiprocessors with a modest number of CPUs cost hundreds of thousands of dollars. Today, multiprocessors with dozens of processors are still quite expensive, but small systems are readily available for a low price. Dell, Gateway, and other companies sell dual-CPU multiprocessors for less than $5,000, and you can purchase a quad-processor system for less than $20,000.

It is possible to write parallel programs for multiprocessors using MPI, but you can often achieve better performance by using a programming language tailored for a shared-memory environment. Recently, OpenMP has emerged as a shared-memory standard. OpenMP is an application programming interface (API) for parallel programming on multiprocessors. It consists of a set of compiler directives and a library of support functions. OpenMP works in conjunction with standard Fortran, C, or C++.

This chapter introduces shared-memory parallel programming using OpenMP. You can use it in two different ways. Perhaps the only parallel computer you have access to is a multiprocessor. In that case, you may prefer to write programs using OpenMP rather than MPI.

On the other hand, you may have access to a multicomputer consisting of many nodes, each of which is a multiprocessor. This is a popular way to build large multicomputers with hundreds or thousands of processors. Consider

these examples (circa 2002):

- IBM's RS/6000 SP system contains up to 512 nodes. Each node can have up to 16 CPUs in it.
- Fujitsu's AP3000 Series supercomputer contains up to 1024 nodes, and each node consists of one or two UltraSPARC processors.
- Dell's High Performance Computing Cluster has up to 64 nodes. Each node is a multiprocessor with two Pentium III CPUs.

In this chapter you'll see how the shared-memory programming model is different from the message-passing model, and you'll learn enough OpenMP compiler directives and functions to be able to parallelize a wide variety of C code segments.

This chapter introduces a powerful set of OpenMP compiler directives:

- `parallel`, which precedes a block of code to be executed in parallel by multiple threads
- `for`, which precedes a `for` loop with independent iterations that may be divided among threads executing in parallel
- `parallel for`, a combination of the `parallel` and `for` directives
- `sections`, which precedes a series of blocks that may be executed in parallel
- `parallel sections`, a combination of the `parallel` and `sections` directives
- `critical`, which precedes a critical section
- `single`, which precedes a code block to be executed by a single thread

You'll also encounter four important OpenMP functions:

- `omp_get_num_procs`, which returns the number of CPUs in the multiprocessor on which this thread is executing
- `omp_get_num_threads`, which returns the number of threads active in the current parallel region
- `omp_get_thread_num`, which returns the thread identification number
- `omp_set_num_threads`, which allows you to fix the number of threads executing the parallel sections of code

17.2 THE SHARED-MEMORY MODEL

The shared-memory model (Figure 17.1) is an abstraction of the generic centralized multiprocessor described in Section 2.4. The underlying hardware is assumed to be a collection of processors, each with access to the same shared memory. Because they have access to the same memory locations, processors can interact and synchronize with each other through shared variables.

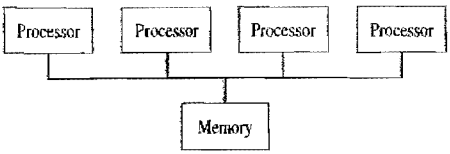


Figure 17.1 The shared-memory model of parallel computation. Processors synchronize and communicate with each other through shared variables.

The standard view of parallelism in a shared-memory program is **fork/join parallelism.** When the program begins execution, only a single thread, called the **master thread,** is active (Figure 17.2). The master thread executes the sequential portions of the algorithm. At those points where parallel operations are required, the master thread forks (creates or awakens) additional threads. The master thread and the created threads work concurrently through the parallel section. At the end of the parallel code the created threads die or are suspended, and the flow of control returns to the single master thread. This is called a join.

A key difference, then, between the shared-memory model and the message-passing model is that in the message-passing model all processes typically remain active throughout the execution of the program, whereas in the shared-memory model the number of active threads is one at the program's start and finish and may change dynamically throughout the execution of the program.

You can view a sequential program as a special case of a shared-memory parallel program: it is simply one with no fork/joins in it. Parallel shared-memory programs range from those with only a single fork/join around a single loop to those in which most of the code segments are executed in parallel. Hence the shared-memory model supports **incremental parallelization,** the process of transforming a sequential program into a parallel program one block of code at a time.

The ability of the shared-memory model to support incremental parallelization is one of its greatest advantages over the message-passing model. It allows you to profile the execution of a sequential program, sort the program blocks according to how much time they consume, consider each block in turn beginning with the most time-consuming, parallelize each block amenable to parallel execution, and stop when the effort required to achieve further performance improvements is not warranted.

Consider, in contrast, message-passing programs. They have no shared memory to hold variables, and the parallel processes are active throughout the execution of the program. Transforming a sequential program into a parallel program is not incremental at all—the chasm must be crossed with one giant leap, rather than many small steps.

In this chapter you'll encounter increasingly complicated blocks of sequential code and learn how to transform them into parallel code sections.

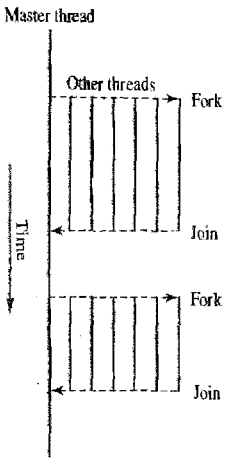


Figure 17.2 The shared-memory model is characterized by fork/join parallelism, in which parallelism comes and goes. At the beginning of execution only a single thread, called the master thread, is active. The master thread executes the serial portions of the program. It forks additional threads to help it execute parallel portions of the program. These threads are deactivated when serial execution resumes.

17.3 PARALLEL `for` LOOPS

Inherently parallel operations are often expressed in C programs as `for` loops. OpenMP makes it easy to indicate when the iterations of a `for` loop may be executed in parallel. For example, consider the following loop, which accounts for a large proportion of the execution time in our MPI implementation of the Sieve of Eratosthenes:

```
for (i = first; i < size; i += prime) marked[i] = 1;
```

Clearly there is no dependence between one iteration of the loop and another. How do we convert it into a parallel loop? In OpenMP we simply indicate to

the compiler that the iterations of a `for` loop may be executed in parallel; the compiler takes care of generating the code that forks/joins threads and **schedules** the iterations, that is, allocates iterations to threads.

17.3.1 `parallel for` Pragma

A compiler directive in C or C++ is called a **pragma.** The word *pragma* is short for "pragmatic information." A pragma is a way to communicate information to the compiler. The information is nonessential in the sense that the compiler may ignore the information and still produce a correct object program. However, the information provided by the pragma can help the compiler optimize the program.

Like other lines that provide information to the preprocessor, a pragma begins with the # character. A pragma in C or C++ has this syntax:

```
#pragma omp <rest of pragma>
```

The first pragma we are going to consider is the `parallel for` pragma. The simplest form of the `parallel for` pragma is:

```
#pragma omp parallel for
```

Putting this line immediately before the `for` loop instructs the compiler to try to parallelize the loop:

```
#pragma omp parallel for
  for (i = first; i < size; i += prime) marked[i] = 1;
```

In order for the compiler to successfully transform the sequential loop into a parallel loop, it must be able to verify that the run-time system will have the information it needs to determine the number of loop iterations when it evaluates the control clause. For this reason the control clause of the `for` loop must have **canonical shape,** as illustrated in Figure 17.3. In addition, the `for` loop must not contain statements that allow the loop to be exited prematurely. Examples include the `break` statement, `return` statement, `exit` statement, and `goto` statements

```
for (index = start; index { <, <=, >=, > } end; { index++, ++index, index--, --index, index += inc, index -= inc, index = index + inc, index = inc + index, index = index - inc } )
```

Figure 17.3 In order to be made parallel, the control clause of a `for` loop must have canonical shape. This figure shows the legal variants. The identifiers *start, end,* and *inc* may be expressions.

to labels outside the loop. The `continue` statement is allowed, however, because its execution does not affect the number of loop iterations.

Our example `for` loop

```
for (i = first; i < size; i += prime) marked[i] = 1;
```

meets these criteria: the control clause has canonical shape, and there are no premature exits in the body of the loop. Hence the compiler can generate code that allows its iterations to execute in parallel.

During parallel execution of the `for` loop, the master thread creates additional threads, and all threads work together to cover the iterations of the loop. Every thread has its own **execution context:** an address space containing all of the variables the thread may access. The execution context includes static variables, dynamically allocated data structures in the heap, and variables on the run-time stack.

The execution context includes its own additional run-time stack, where the frames for functions it invokes are kept. Other variables may either be shared or private. A **shared variable** has the same address in the execution context of every thread. All threads have access to shared variables. A **private variable** has a different address in the execution context of every thread. A thread can access its own private variables, but cannot access the private variable of another thread.

In the case of the `parallel for` pragma, variables are by default shared, with the exception that the loop index variable is private.

Figure 17.4 illustrates shared and private variables. In this example the iterations of the `for` loop are being divided among two threads. The loop index `i` is a private variable—each thread has its own copy. The remaining variables `b` and `ptr`, as well as data allocated on the heap, are shared.

How does the run-time system know how many threads to create? The value of an environment variable called `OMP_NUM_THREADS` provides a default number of threads for parallel sections of code. In Unix you can use the `printenv` command to inspect the value of this variable and the `setenv` command to modify its value.

```
int main (int argc, char* argv[])
{
  int b[3];
  char* cptr ;
  int i;

  cptr = malloc (1);
#pragma omp parallel for
  for (i=0; i<3; i++)
    b[i]=i;
```

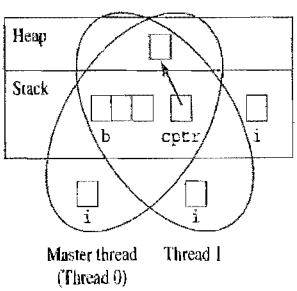


Figure 17.4 During parallel execution of the `for` loop, index `i` is a private variable, while `b`, `cptr`, and heap data are shared.

Another strategy is to set the number of threads equal to the number of multiprocessor CPUs. Let's explore the OpenMP functions that enable us to do this.

17.3.2 Function `omp_get_num_procs`

Function `omp_get_num_procs` returns the number of physical processors available for use by the parallel program. Here is the function header:

```
int omp_get_num_procs (void)
```

The integer returned by this function may be less than the total number of physical processors in the multiprocessor, depending on how the run-time system gives processes access to processors.

17.3.3 Function `omp_set_num_threads`

Function `omp_set_num_threads` uses the parameter value to set the number of threads to be active in parallel sections of code. It has this function header:

```
void omp_set_num_threads (int t)
```

Since this function may be called at multiple points in a program, you have the ability to tailor the level of parallelism to the grain size or other characteristics of the code block.

Setting the number of threads equal to the number of available CPUs is straightforward:

```
int t;
...
t = omp_get_num_procs();
omp_set_num_threads(t);
```

17.4 DECLARING PRIVATE VARIABLES

For our second example, let's look at slightly more complicated loop structure. Here is the computational heart of our MPI implementation of Floyd's algorithm:

```
for (i = 0; i < BLOCK_SIZE(id,p,n); i++)
   for (j = 0; j < n; j++)
      a[i][j] = MIN(a[i][j], a[i][k] + tmp[j]);
```

In our earlier analysis of this algorithm, we determined that either loop could be executed in parallel. Which one should we choose? If we parallelize the inner loop, then the program will fork and join threads for every iteration of the outer loop. The fork/join overhead may very well be greater than the time saved by dividing the execution of the n iterations of the inner loop among multiple threads. On the other hand, if we parallelize the outer loop, the program only incurs the fork/join overhead once.

Grain size is the number of computations performed between communication or synchronization steps. In general, increasing grain size improves the performance of a parallel program. Making the outer loop parallel results in larger grain size. It is the option we choose.

It's easy enough to direct the compiler to execute the iterations of the loop indexed by `i` in parallel. However, we need to pay attention to the variables accessed by the threads. By default, all variables are shared except loop index `i`. That makes it easy for threads to communicate with each other, but it can also cause problems.

Consider what happens when multiple threads try to execute different iterations of the `i` loop in parallel. We want every thread to work through `n` values of `j` for each iteration of the `i` loop. However, all of the threads try to initialize and increment the same shared variable `j`—meaning that there is a good chance threads will not execute all `n` iterations.

The solution is clear—we need to make `j` a private variable, too.

17.4.1 `private` Clause

A clause is an optional, additional component to a pragma. The `private` clause directs the compiler to make one or more variables private. It has this syntax:

`private (`*<variable list>*`)`

The directive tells the compiler to allocate a private copy of the variable for each thread executing the block of code the pragma precedes. In our case, we are making a `for` loop parallel. The private copies of variable `j` will be accessible only inside the `for` loop. The values are undefined on loop entry and exit.

Using the `private` clause, a correct OpenMP implementation of the doubly nested loops is

```
#pragma omp parallel for private(j)
   for (i = 0; i < BLOCK_SIZE(id,p,n); i++)
      for (j = 0; j < n; j++)
         a[i][j] = MIN(a[i][j], a[i][k] + tmp[j]);
```

Even if `j` had a previously assigned value before entering the parallel `for` loop, none of the threads can access that value. Similarly, whatever values the threads assign to `j` during execution of the parallel `for` loop, the value of the shared `j` will not be affected. Put another way, by default the value of a private variable is undefined when the parallel construct is entered, and the value is also undefined when the construct is exited.

The default condition of private variables (undefined at loop entry and exit) reduces execution time by eliminating unnecessary copying between shared variables and their private variable counterparts.

17.4.2 `firstprivate` Clause

Sometimes we want a private variable to inherit the value of the shared variable. Consider, for example, the following code segment:

```
x[0] = complex_function();
for (i = 0; i < n; i++) {
   for (j = 1 ; j < 4; j++)
      x[j] = g(i, x[j-1]);
   answer[i] = x[1] - x[3];
}
```

Assuming function g has no side effects, we may execute every iteration of the outer loop in parallel, as long as we make x a private variable. However, `x[0]` is initialized before the outer `for` loop and referenced in the first iteration of the inner loop. It is impractical to move the initialization of `x[0]` inside the outer `for` loop, because it is too time-consuming. Instead, we want each thread's private copy of array element `x[0]` to inherit the value the shared variable was assigned in the master thread.

The `firstprivate` clause, with syntax

`firstprivate (`*<variable list>*`)`

does just that. It directs the compiler to create private variables having initial values identical to the value of the variable controlled by the master thread as the loop is entered.

Here is the correct way to code the parallel loop:

```
x[0] = complex_function();
#pragma omp parallel for private(j) firstprivate(x)
for (i = 0; i < n; i++) {
   for (j = 1 ; j < 4; j++)
      x[j] = g(i, x[j-1]);
   answer[i] = x[1] - x[3];
}
```

Note that the values of the variables in the `firstprivate` list are initialized once per thread, not once per iteration. If a thread executes multiple iterations of the parallel loop and modifies the value of one of these variables in an iteration, then subsequent iterations referencing the variable will get the modified value, not the original value.

17.4.3 `lastprivate` Clause

The **sequentially last iteration** of a loop is the iteration that occurs last when the loop is executed sequentially. The `lastprivate` clause directs the compiler to generate code at the end of the parallel `for` loop that copies back to the master

thread's copy of a variable the private copy of the variable from the thread that executed the sequentially last iteration of the loop.

For example, suppose we were parallelizing the following piece of code:

```
for (i = 0; i < n; i++) {
   x[0] = 1.0;
   for (j = 1 ; j < 4; j++)
      x[j] = x[j-1] * (i+1);
   sum_of_powers[i] = x[0] + x[1] + x[2] + x[3];
}
n_cubed = x[3];
```

In the sequentially last iteration of the loop, `x[3]` gets assigned the value n^3. In order to have this value accessible outside the parallel `for` loop, we must declare `x` to be a `lastprivate` variable. Here is the correct parallel version of the loop:

```
#pragma omp parallel for private(j) lastprivate(x)
for (i = 0; i < n; i++) {
   x[0] = 1.0;
   for (j = 1 ; j < 4; j++)
      x[j] = x[j-1] * (i+1);
   sum_of_powers[i] = x[0] + x[1] + x[2] + x[3];
}
n_cubed = x[3];
```

A `parallel for` pragma may contain both `firstprivate` and `lastprivate` clauses. If the same pragma has both of these clauses, the clauses may have none, some, or all of the variables in common.

17.5 CRITICAL SECTIONS

Let's consider part of a C program that estimates the value of π using a form of numerical integration called the rectangle rule:

```
double area, pi, x;
int i, n;
...
area = 0.0;
for (i = 0; i < n; i++) {
   x = (i+0.5)/n;
   area += 4.0/(1.0 + x*x);
}
pi = area / n;
```

Unlike the `for` loops we have already considered, the iterations of this loop are not independent of each other. Each iteration of the loop reads and updates the value of `area`. If we simply parallelize the loop:

```
   double area, pi, x;
   int i, n;
   ...
   area = 0.0;
#pragma omp parallel for private(x)
   for (i = 0; i < n; i++) {
      x = (i+0.5)/n;
      area += 4.0/(1.0 + x*x);   /* Race condition! */
   }
   pi = area / n;
```

we may not end up with the correct answer, because the execution of the assignment statement is not an atomic (indivisible) operation. This sets up a **race condition,** in which the computation exhibits nondeterministic behavior when performed by multiple threads accessing a shared variable.

See Figure 17.5. Suppose thread A and thread B are concurrently executing different iterations of the loop. Thread A reads the current value of `area` and computes the sum

```
area + 4.0/(1.0 + x*x)
```

Before it can write the value of the sum back to `area`, thread B reads the current value of `area`. Thread A updates the value of `area` with the sum. Thread B computes its sum and writes back the value. Now the value of `area` is incorrect.

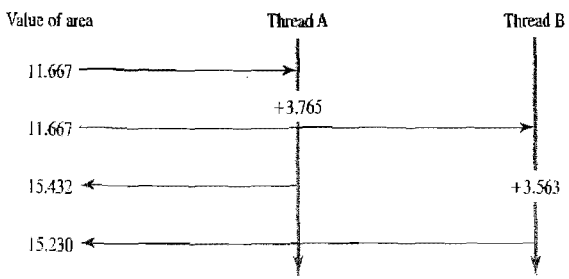


Figure 17.5 Example of a race condition. Each thread is adding a value to `area`. However, Thread B retrieves the original value of `area` before Thread A can write the new value. Hence the final value of `area` is incorrect. If Thread B had read the value of `area` after Thread A had updated it, then the final value of `area` would have been correct. In short, the absence of a critical section can lead to nondeterministic execution.

The assignment statement that reads and updates `area` must be put in a **critical section**—a portion of code that only one thread at a time may execute.

17.5.1 `critical` Pragma

We can denote a critical section in OpenMP by putting the pragma

```
#pragma omp critical
```

in front of a block of C code. (A single statement is a trivial example of a code block.) This pragma directs the compiler to enforce mutual exclusion among the threads trying to execute the block of code.

After adding the critical pragma, our code looks like this:

```
   double area, pi, x;
   int i, n;
   ...
   area = 0.0;
#pragma omp parallel for private(x)
   for (i = 0; i < n; i++) {
      x = (i+0.5)/n;
#pragma omp critical
      area += 4.0/(1.0 + x*x);
   }
   pi = area / n;
```

At this point our C/OpenMP code segment will produce the correct result. The iterations of the `for` loop are divided among the threads, and only one thread at a time may execute the assignment statement that updates the value of `area`. However, this code segment will exhibit poor speedup. Since it admits only one thread at a time, the critical section is a piece of sequential code inside the `for` loop. The time to execute this statement is nontrivial. Hence by Amdahl's Law we know the critical section will put a low ceiling on the speedup achievable by parallelizing the `for` loop.

Of course, what we are really trying to do is perform a sum-reduction of `n` values. In the next section we'll learn an efficient way to code a reduction.

17.6 REDUCTIONS

Reductions are so common that OpenMP allows us to add a reduction clause to our `parallel for` pragma. All we have to do is specify the reduction operation and the **reduction variable**, and OpenMP will take care of the details, such as storing partial sums in private variables and then adding the partial sums to the shared variable after the loop.

The reduction clause has this syntax:

`reduction(<op>:<variable>)`

where *<op>* is one of the reduction operators shown in Table 17.1 and *<variable>* is the name of the shared variable that will end up with the result of the reduction.

Here is an implementation of the π-finding code with the reduction clause replacing the critical section:

```
   double area, pi, x;
   int i, n;
   ...
   area = 0.0;
#pragma omp parallel for private(x) reduction(+:area)
   for (i = 0; i < n; i++) {
      x = (i+0.5)/n;
      area += 4.0/(1.0 + x*x);
   }
   pi = area / n;
```

Table 17.2 compares our two implementations of the rectangle rule to compute π. We set $n = 100{,}000$ and execute the programs on a Sun Enterprise Server 4000. The implementation that uses the `reduction` clause is clearly superior to the one using the `critical` pragma. It is faster when only a single thread is active, and the execution time improves when additional threads are added.

Table 17.1 OpenMP reduction operators for C and C++.

Operator	Meaning	Allowable types	Initial value
+	Sum	float, int	0
*	Product	float, int	1
&	Bitwise and	int	all bits 1
|	Bitwise or	int	0
^	Bitwise exclusive or	int	0
&&	Logical and	int	1
||	Logical or	int	0

Table 17.2 Execution times on a Sun Enterprise Server 4000 of two programs that compute π using the rectangle rule.

	Execution time of program (sec)	
Threads	Using `critical` pragma	Using `reduction` clause
1	0.0780	0.0273
2	0.1510	0.0146
3	0.3400	0.0105
4	0.3608	0.0086
5	0.4710	0.0076

17.7 PERFORMANCE IMPROVEMENTS

Sometimes transforming a sequential `for` loop into a parallel `for` loop can actually increase a program's execution time. In this section we'll look at three ways of improving the performance of parallel loops.

17.7.1 Inverting Loops

Consider the following code segment:

```
for (i = 1; i < m; i++)
   for (j = 0; j < n; j++)
      a[i][j] = 2 * a[i-1][j];
```

We can draw a data dependence diagram to help us understand the data dependences in this code. The diagram appears in Figure 17.6. We see that two rows may not be updated simultaneously, because there are data dependences between rows. However, the columns may be updated simultaneously. This means the loop indexed by `j` may be executed in parallel, but not the loop indexed by `i`.

If we insert a `parallel for` pragma before the inner loop, the resulting parallel program will execute correctly, but it may not exhibit good performance, because it will require `m-1` fork/join steps, one per iteration of the outer loop.

However, if we invert the loops:

```
#pragma parallel for private(i)
for (j = 0; j < n; j++)
   for (i = 1; i < m; i++)
      a[i][j] = 2 * a[i-1][j];
```

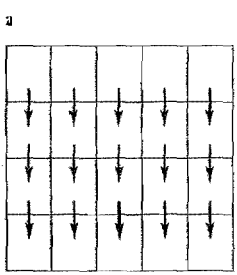


Figure 17.6 Data dependence diagram for a particular pair of nested loops shows that while columns may be updated simultaneously, rows cannot.

only a single fork/join step is required (surrounding the outer loop). The data dependences have not changed; the iterations of the loop indexed by `j` are still independent of each other. In this respect we have definitely improved the code.

However, we must always be cognizant of how code transformations affect the cache hit rate. In this case, each thread is now working through columns of `a`, rather than rows. Since C matrices are stored in row-major order, inverting loops may lower the cache hit rate, depending upon `m`, `n`, the number of active threads, and the architecture of the underlying system.

17.7.2 Conditionally Executing Loops

If a loop does not have enough iterations, the time spent forking and joining threads may exceed the time saved by dividing the loop iterations among multiple threads. Consider, for example, the parallel implementation of the rectangle rule we examined earlier:

```
   area = 0.0;
#pragma omp parallel for private(x) reduction (+:area)
   for (i = 0; i < n; i++) {
      x = (i + 0.5)/n;
      area += 4.0 / (1.0 + x * x);
   }
   pi = area / n;
```

Table 17.3 reveals the average execution time of this program segment on a Sun Enterprise Server 4000, for various values of `n` and various numbers of threads. As you can see, when `n` is 100, the sequential execution time is so small that adding threads only increases overall execution time. When `n` is 100,000, the parallel program executing on four threads achieves a speedup of 3.16 over the sequential program.

The `if` clause gives us the ability to direct the compiler to insert code that determines at run-time whether the loop should be executed in parallel or

Table 17.3 Execution time on a Sun Enterprise Server 4000 of a parallel C program that computes π using the rectangle rule, as a function of number of rectangles and number of threads.

	Execution time (msec)	
Threads	$n = 100$	$n = 100{,}000$
1	0.964	27.288
2	1.436	14.598
3	1.732	10.506
4	1.990	8.648

sequentially. The clause has this syntax:

`if (`*<scalar expression>*`)`

If the scalar expression evaluates to true, the loop will be executed in parallel. Otherwise, it will be executed serially.

For example, here is how we could add an `if` clause to the `parallel for` pragma in the parallel program computing π using the rectangle rule:

```
#pragma omp parallel for private(x) reduction(+:area) if(n > 5000)
   for (i = 0; i < n; i++) {
   ...
```

In this case loop iterations will be divided among multiple threads only if $n > 5{,}000$.

17.7.3 Scheduling Loops

In some loops the time needed to execute different loop iterations varies considerably. For example, consider the following doubly nested loop that initializes an upper triangular matrix:

```
for (i = 0; i < n; i++)
   for (j = i; j < n; j++)
      a[i][j] = alpha_omega(i,j);
```

Assuming there are no data dependences among iterations, we would prefer to execute the outermost loop in parallel in order to minimize fork/join overhead. If every call to function `alpha_omega` takes the same amount of time, then the first iteration of the outermost loop (when `i` equals 0) requires `n` times more work than the last iteration (when `i` equals `n-1`). Inverting the two loops will not remedy the imbalance.

Suppose these n iterations are being executed on t threads. If each thread is assigned a contiguous block of either $\lceil n/t \rceil$ or $\lfloor n/t \rfloor$ threads, the parallel loop execution will have poor efficiency, because some threads will complete their share of the iterations much faster than others.

The `schedule` clause allows us to specify how the iterations of a loop should be scheduled, that is, allocated to threads. In a **static schedule,** all iterations are allocated to threads before they execute any loop iterations. In a **dynamic schedule,** only some of the iterations are allocated to threads at the beginning of the loop's execution. Threads that complete their iterations are then eligible to get additional work. The allocation process continues until all of the iterations have been distributed to threads. Static schedules have low overhead but may exhibit high load imbalance. Dynamic schedules have higher overhead but can reduce load imbalance.

In both static and dynamic schedules, contiguous ranges of iterations called **chunks** are assigned to threads. Increasing the chunk size can reduce overhead

and increase the cache hit rate. Reducing the chunk size can allow finer balancing of workloads.

The schedule clause has this syntax:

```
schedule(<type> [, <chunk>])
```

In other words, the schedule type is required, but the chunk size is optional. With these two parameters it's easy to describe a wide variety of schedules:

- `schedule(static)`: A static allocation of about `n/t` contiguous iterations to each thread.
- `schedule(static,C)`: An interleaved allocation of chunks to tasks. Each chunk contains C contiguous iterations.
- `schedule(dynamic)`: Iterations are dynamically allocated, one at a time, to threads.
- `schedule(dynamic,C)`: A dynamic allocation of C iterations at a time to the tasks.
- `schedule(guided,C)`: A dynamic allocation of iterations to tasks using the guided self-scheduling heuristic. **Guided self-scheduling** begins by allocating a large chunk size to each task and responds to further requests for chunks by allocating chunks of decreasing size. The size of the chunks decreases exponentially to a minimum chunk size of C.
- `schedule(guided)`: Guided self-scheduling with a minimum chunk size of 1.
- `schedule(runtime)`: The schedule type is chosen at run-time based on the value of the environment variable `OMP_SCHEDULE`. For example, the Unix command

  ```
  setenv OMP_SCHEDULE "static,1"
  ```

  would set the run-time schedule to be an interleaved allocation.

When the `schedule` clause is not included in the `parallel for` pragma, most run-time systems default to a simple static scheduling of consecutive loop iterations to tasks.

Going back to our original example, the run-time of any particular iteration of the outermost `for` loop is predictable. An interleaved allocation of loop iterations balances the workload of the threads:

```
#pragma omp parallel for private(j) schedule(static,1)
for (i = 0; i < n; i++)
   for (j = i; j < n; j++)
      a[i][j] = alpha_omega(i,j);
```

Increasing the chunk size from 1 could improve the cache hit rate at the expense of increasing the load imbalance. The best value for the chunk size is system-dependent.

17.8 MORE GENERAL DATA PARALLELISM

To this point we have focused on the parallelization of simple `for` loops. They are perhaps the most common opportunity for parallelism, particularly in programs that have already been written in MPL However, we should not ignore other opportunities for concurrency. In this section we look at two examples of data parallelism outside simple `for` loops.

First let's consider an algorithm to process a linked list of tasks. We considered a similar algorithm when we designed a solution to the document classification problem in Chapter 9. In that design, we assumed a message-passing model. Because that model has no shared memory, we gave a single process, which we called the manager, responsibility for maintaining the entire list of tasks. Worker tasks sent messages to the manager when they were ready to process another task.

In contrast, the shared-memory model allows every thread to access the same "to-do" list, so there is no need for a separate manager thread.

The following code segments are part of a program that processes work stored in a singly linked to-do list (see Figure 17.7):

```
int main (int argc, char argv[])
{
   struct job_struct job_ptr;
   struct task_struct task_ptr;

   ...
   task_ptr = get_next_task (&job_ptr);
   while (task_ptr != NULL) {
      complete_task (task_ptr);
      task_ptr = get_next_task (&job_ptr);
   }
   ...
}

char get_next_task(struct job_struct job_ptr) {
   struct task_struct answer;
   if (job_ptr == NULL) answer = NULL;
   else {
      answer = (job_ptr)->task;
      job_ptr = (job_ptr)->next;
   }
   return answer;
}
```

How would we like this algorithm to execute in parallel? We want every thread to do the same thing: repeatedly take the next task from the list and complete it, until there are no more tasks to do. We need to ensure that no two threads take the same task from the list. In other words, it is important to execute function `get_next_task` atomically.

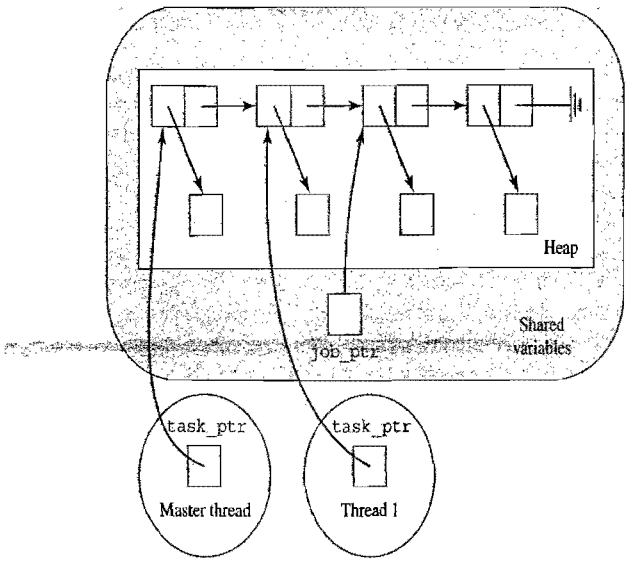


Figure 17.7 Two threads work their way through a singly linked "to do" list. Variable `job_ptr` must be shared, while `task_ptr` must be a private variable.

17.8.1 `parallel` Pragma

The parallel pragma precedes a block of code that should be executed by all of the threads. It has this syntax:

```
#pragma omp parallel
```

If the code we want executed in parallel is not a simple statement (such as an assignment statement, `if` statement, or `for` loop) we can use curly braces to create a block of code from a statement group.

Note that unlike the `parallel for` pragma, which divided the iterations of the `for` loop among the active threads, the execution of the code block after the parallel program is replicated among the threads. Our section of function `main` now looks like this:

```
int main (int argc, char argv[])
{
   struct job_struct job_ptr;
   struct task_struct task_ptr;

#pragma omp parallel private (task_ptr)
   {
      task_ptr = get_next_task (&job_ptr);
```

```
      while (task_ptr != NULL) {
         complete_task (task_ptr);
         task_ptr = get_next_task (&job_ptr);
      }
   }
}
```

Now we need to ensure function `get_next_task` executes atomically. Otherwise, allowing two threads to execute function `get_next_task` simultaneously may result in more than one thread returning from the function with the same value of `task_ptr`.

We use the `critical` pragma to ensure mutually exclusive execution of this critical section of code. Here is the rewritten function `get_next_task`:

```
char get_next_task(struct job_struct job_ptr) {
   struct task_struct answer;

#pragma omp critical
   {
      if (job_ptr == NULL) answer = NULL;
      else {
         answer = (job_ptr)->task;
         job_ptr = (job_ptr)->next;
      }
   }
   return answer;
}
```

17.8.2 Function `omp_get_thread_num`

Earlier in this chapter we computed π using the rectangle rule. In Chapter 10 we computed π using the Monte Carlo method. The idea, illustrated in Figure 17.8, is to generate pairs of points in the unit square (where each coordinate varies between 0 and 1). We count the fraction of points inside the circle (those points for which $x^2 + y^2 \leq 1$). The expected value of this fraction is $\pi/4$; hence multiplying the fraction by 4 gives an estimate of π.

Here is the C code implementing the algorithm:

```
int      count;          /* Points inside unit circle */
unsigned short xi[3];    /* Random number seed */
int      i;
int      samples;        /* Points to generate */
double   x, y;           /* Coordinates of point */
samples = atoi (argv[1]);
xi[0] = atoi (argv[2]);
xi[1] = atoi (argv[3]);
```

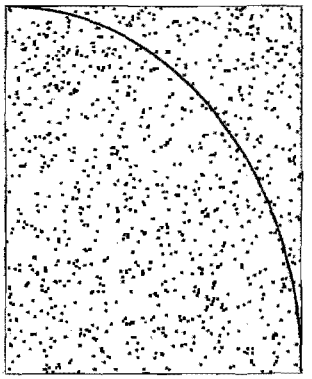


Figure 17.8 Example of a Monte Carlo algorithm to compute π. In this example we have generated 1,000 pairs from a uniform distribution between 0 and 1. Since 773 pairs are inside the unit circle, our estimate of π is 4(773/1000), or 3.092.

```
xi[2] = atoi (argv[4]);
count = 0;
for (i = 0; i < samples; i++) {
   x = erand48(xi);
   y = erand48(xi);
   if (x*x+y*y <= 1.0) count++;
}
printf ("Estimate of pi: %7.5f\n", 4.0*count/samples);
```

If we want to speed the execution of the program by using multiple threads, we must ensure that each thread is generating a different stream of random numbers. Otherwise, each thread would generate the same sequence of (x, y) pairs, and there would be no increase in the precision of the answer through the use of parallelism. Hence `xi` must be a private variable, and must find some way for each thread to initialize array `xi` with unique values. That means we need to have some way of distinguishing threads.

In OpenMP every thread on a multiprocessor has a unique identification number. We can retrieve this number using the function `omp_get_thread_num`, which has this header:

```
int omp_get_thread_num (void)
```

If there are t active threads, the thread identification numbers are integers ranging from 0 through $t - 1$. The master thread always has identification number 0.

Assigning the thread identification number to `xi[2]` ensures each thread has a different random number seed.

17.8.3 Function `omp_get_num_threads`

In order to divide the iterations among the threads, we must know the number of active threads. Function `omp_get_num_threads`, with this header

```
int omp_get_num_threads(void)
```

returns the number of threads active in the current parallel region. We can use this information, as well as the thread identification number, to divide the iterations among the threads.

Each thread will accumulate its count of points inside the circle in a private variable. When each thread completes the `for` loop, it will add its subtotal to `count` inside a critical section.

The OpenMP implementation of the Monte Carlo π-finding algorithm appears in Figure 17.9.

17.8.4 `for` Pragma

The `parallel` pragma can also come in handy when parallelizing `for` loops. Consider this doubly nested loop:

```
for (i = 0; i < m; i++) {
   low = a[i];
   high = b[i];
   if (low > high) {
      printf ("Exiting during iteration %d\n", i);
      break;
   }
   for (j = low; j < high; j++)
      c[j] = (c[j] - a[i])/b[i];
}
```

We cannot execute the iterations of the outer loop in parallel, because it contains a `break` statement. If we put a `parallel for` pragma before the loop indexed by `j`, there will be a fork/join step for every iteration of the outer loop. We would like to avoid this overhead. Previously, we showed how inverting `for` loops could solve this problem, but that approach doesn't work here because of the data dependences.

If we put the `parallel` pragma immediately in front of the loop indexed by `i`, then we'll only have a single fork/join. The default behavior is that *every* thread executes *all* of the code inside the block. Of course, we want the threads to divide up the iterations of the inner loop. The `for` pragma directs the compiler to do just that:

```
#pragma omp for
```

```
/*
 * OpenMP implementation of Monte Carlo pi-finding algorithm
 */

#include <stdio.h>
int main (int argc, char *argv[]){
   int      count;          /* Points inside unit circle */
   int      i;
   int      local_count;    /* This thread's subtotal */
   int      samples;        /* Points to generate */
   unsigned short xi[3];    /* Random number seed */
   int      t;              /* Number of threads */
   int      tid;            /* Thread id */
   double   x, y;           /* Coordinates of point */

   /* Number of points and number of threads are
      command-line arguments */

   samples = atoi(argv[1]);
   omp_set_num_threads (atoi(argv[2]));

   count = 0;
#pragma omp parallel private(xi,t,i,x,y,local_count)
   {
      local_count = 0;
      xi[0] = atoi(argv[3]);
      xi[1] = atoi(argv[4]);
      xi[2] = tid = omp_get_thread_num();
      t = omp_get_num_threads();

      for (i = tid; i < samples; i += t) {
         x = erand48(xi);
         y = erand48(xi);
         if (x*x+y*y <= 1.0) local_count++;
      }
#pragma omp critical
      count += local_count;
   }
   printf ("Estimate of pi: %7.5f\n", 4.0*count/samples);
}
```

Figure 17.9 This C/OpenMP program uses the Monte Carlo method to compute π.

With these pragmas added, our code segment looks like this:

```
#pragma omp parallel private(i,j)
for (i = 0; i < m; i++) {
   low = a[i];
   high = b[i];
   if (low > high) {
      printf ("Exiting during iteration %d\n", i);
      break;
   }
```

```
#pragma omp for
   for (j = low; j < high; j++)
      c[j] = (c[j] - a[i])/b[i];
}
```

Our work is not yet complete, however.

17.8.5 `single` Pragma

We have parallelized the execution of the loop indexed by `j`. What about the other code inside the outer loop? We certainly don't want to see the error message more than once.

The `single` pragma tells the compiler that only a single thread should execute the block of code the pragma precedes. Its syntax is:

```
#pragma omp single
```

Adding the `single` pragma to the code block, we now have:

```
#pragma omp parallel private(i,j)
for (i = 0; i < m; i++) {
   low = a[i];
   high = b[i];
   if (low > high) {
#pragma omp single
      printf ("Exiting during iteration %d\n", i);
      break;
   }
#pragma omp for
   for (j = low; j < high; j++)
      c[j] = (c[j] - a[i])/b[i];
}
```

The code block now executes correctly, but we can improve its performance.

17.8.6 `nowait` Clause

The compiler puts a barrier synchronization at the end of every `parallel for` statement. In the example we have been considering, this barrier is necessary, because we need to ensure that every thread has completed one iteration of the loop indexed by `i` before any thread begins the next iteration. Otherwise, a thread might change the value of `low` or `high`, altering the number of iterations of the `j` loop performed by another thread.

On the other hand, if we make `low` and `high` private variables, there is no need for the barrier at the end of the loop indexed by `j`. The `nowait` clause, added to a `parallel for` pragma, tells the compiler to omit the barrier synchronization at the end of the parallel `for` loop.

After making low and high private and adding the nowait clause, our final version of our example code segment is:

```
#pragma omp parallel private(i,j,low,high)
for (i = 0; i < m; i++) {
   low = a[i];
   high = b[i];
   if (low > high) {
#pragma omp single
      printf ("Exiting during iteration %d\n", i);
      break;
   }
#pragma omp for nowait
   for (j = low; j < high; j++)
      c[j] = (c[j] - a[i])/b[i];
}
```

17.9 FUNCTIONAL PARALLELISM

To this point we have focused entirely on exploiting data parallelism. Another source of concurrency is functional parallelism. OpenMP allows us to assign different threads to different portions of code.

Consider, for example, the following code segment:

```
v = alpha();
w = beta();
x = gamma(v, w);
y = delta();
printf ("%6.2f\n", epsilon(x,y));
```

If all of the functions are side-effect free, we can represent the data dependences as shown in Figure 17.10. Clearly functions alpha, beta, and delta may be executed in parallel. If we execute these functions concurrently, there is no more

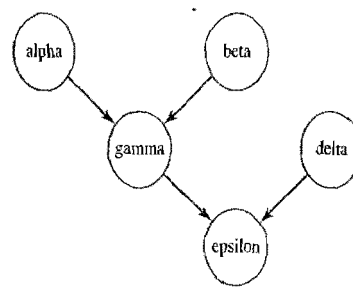


Figure 17.10 Data dependence diagram for code segment of Section 17.9.

functional parallelism to exploit, because function gamma must be called after functions alpha and beta and before function epsilon.

17.9.1 parallel sections Pragma

The parallel sections pragma precedes a block of k blocks of code that may be executed concurrently by k threads. It has this syntax:

```
#pragma omp parallel sections
```

17.9.2 section Pragma

The section pragma precedes each block of code within the encompassing block preceded by the parallel sections pragma. (The section pragma may be omitted for the first parallel section after the parallel sections pragma.)

In the example we considered, the calls to functions alpha, beta, and delta could be evaluated concurrently. In our parallelization of this code segment, we use curly braces to create a block of code containing these three assignment statements. (Recall that an assignment statement is a trivial example of a code block. Hence a block containing three assignment statements is a block of three blocks of code.)

```
#pragma omp parallel sections
   {
#pragma omp section      /* This pragma optional */
      v = alpha();
#pragma omp section
      w = beta();
#pragma omp section
      y = delta();
   }
   x = gamma(v, w);
   printf ("%6.2f\n", epsilon(x,y));
```

Note that we reordered the assignment statements to bring together the three that could be executed in parallel.

17.9.3 sections Pragma

Let's take another look at the data dependence diagram of Figure 17.10. There is a second way to exploit functional parallelism in this code segment. As we noted earlier, if we execute functions alpha, beta, and delta in parallel, there are no further opportunities for functional parallelism. However, if we execute only functions alpha and beta in parallel, then after they return we may execute functions gamma and delta in parallel.

In this design we have two different parallel sections, one following the other. We can reduce fork/join costs by putting all four assignment statements in a single

block preceded by the `parallel` pragma, then using the `sections` pragma to identify the first and second pairs of functions that may execute in parallel.

The `sections` pragma with syntax

```
#pragma omp sections
```

appears inside a `parallel` block of code. It has exactly the same meaning as the `parallel sections` pragma we have already described.

Here is another way to express functional parallelism in the code segment we have been considering, using the `sections` pragma:

```
#pragma omp parallel
   {
   #pragma omp sections
      {
      #pragma omp section      /* This pragma optional */
         v = alpha();
      #pragma omp section
         w = beta();
      }
   #pragma omp sections
      {
      #pragma omp section     /* This pragma optional */
         x = gamma(v, w);
      #pragma omp section
         y = delta();
      }
   }
   printf ("%6.2f\n", epsilon(x,y));
```

In one respect this solution is better than the first one we presented, because it has two parallel sections of code, each requiring two threads. Our first solution has only a single parallel section of code that required three threads. If only two processors are available, the second section of code could result in higher efficiency. Whether or not that is the case depends upon the execution times of the individual functions.

17.10 SUMMARY

OpenMP is an API for shared-memory parallel programming. The shared-memory model relies upon fork/join parallelism. You can envision the execution of a shared-memory program as periods of sequential execution alternating with periods of parallel execution. A master thread executes all of the sequential code. When it reaches a parallel code segment, it forks other threads. The threads communicate with each other via shared variables. At the end of the parallel code segment, these threads synchronize, rejoining the master thread.

This chapter has introduced OpenMP pragmas and clauses that can be used to transform a sequential C program into one that runs in parallel on a multiprocessor. First we considered the parallelization of `for` loops. In C programs data parallelism is often expressed in the form of `for` loops. We use the `parallel for` pragma to indicate to the compiler those loops whose iterations may be performed in parallel. There are certain restrictions on `for` loops that may be executed in parallel. The control clause must be simple, so that the run-time system can determine, before the loop executes, how many iterations it will have. The loop cannot have a `break` statement, `goto` statement, or another statement that allows early loop termination.

We also discussed how to take advantage of functional parallelism through the use of the `parallel sections` pragma. This pragma precedes a block of blocks of code, where each of the inner blocks, or sections, represents an independent task that may be performed in parallel with the other sections.

The `parallel` pragma precedes a block of code that should be executed in parallel by all threads. When all threads execute the same code, the result is SPMD-style parallel execution similar to that exhibited by many of our programs using MPI. A `for` pragma or a `sections` pragma may appear inside the block of code marked with a `parallel` pragma, allowing the compiler to exploit data or functional parallelism.

We also use pragmas to point out areas within parallel sections that must be executed sequentially. The `critical` pragma indicates a block of code forming a critical section where mutual exclusion must be enforced. The `single` pragma indicates a block of code that should only be executed by one of the threads.

We can convey additional information to the compiler by adding clauses to pragmas. The `private` clause gives each thread its own copy of the listed variables. Values can be copies between the original variable and private variables using the `firstprivate` and/or the `lastprivate` clauses. The `reduction` clause allows the compiler to generate efficient code for reduction operations happening inside a parallel loop. The `schedule` clause lets you specify the way loop iterations are allocated to tasks. The `if` clause allows the system to determine at run-time if a construct should be executed sequentially or by multiple threads. The `nowait` clause eliminates the barrier synchronization at the end of the parallel construct.

While we have introduced clauses in the context of particular pragmas, most clauses can be applied to most pragmas. Table 17.4 lists which of the clauses we have introduced in this chapter may be attached to which pragmas.

We have examined various ways in which the performance of parallel `for` loops can be enhanced. The strategies are inverting loops, conditionally parallelizing loops, and changing the way in which loop iterations are scheduled.

Table 17.5 compares OpenMP with MPI. Both programming environments can be used to program multiprocessors. MPI is suitable for programming multicomputers. Since OpenMP has shared variables, OpenMP is not appropriate for generic multicomputers in which there is no shared memory. MPI also makes it easier for the programmer to take control of the memory hierarchy. On the

Table 17.4 This table summarizes which clauses may be attached to which pragmas.

Pragma	Clauses allowed
`critical`	*None*
`for`	`firstprivate, lastprivate, nowait, private, reduction, schedule`
`parallel`	`firstprivate, if, lastprivate, private, reduction`
`parallel for`	`firstprivate, if, lastprivate, private, reduction, schedule`
`parallel sections`	`firstprivate, if, lastprivate, private, reduction`
`sections`	`firstprivate, lastprivate, nowait, private, reduction`
`single`	`firstprivate, nowait, private`

Note: OpenMP has additional clauses not introduced in this chapter.

Table 17.5 Comparison of OpenMP and MPI.

Characteristic	OpenMP	MPI
Suitable for multiprocessors	Yes	Yes
Suitable for multicomputers	No	Yes
Supports incremental parallelization	Yes	No
Minimal extra code	Yes	No
Explicit control of memory hierarchy	No	Yes

other hand, OpenMP has the significant advantage of allowing programs to be incrementally parallelized. In addition, unlike programs using MPI, which often are much longer than their sequential counterparts, programs using OpenMP are usually not much longer than the sequential codes they displace.

17.11 KEY TERMS

canonical shape
chunk
clause
critical section
dynamic schedule
execution context
fork/join parallelism
grain size
guided self-scheduling
incremental parallelization
master thread
pragma
`private` clause
private variable
race condition
reduction variable
schedule
sequentially last iteration
shared variable
static schedule

17.12 BIBLIOGRAPHIC NOTES

The URL for the official OpenMP Web site is `www.OpenMP.org`. You can download the official OpenMP specifications for the C/C++ and Fortran versions of OpenMP from this site.

Parallel Programming in OpenMP by Chandra et al. is an excellent introduction to this shared-memory application programming interface [16]. It provides broader and deeper coverage of the features of OpenMP. It also discusses performance tuning of OpenMP codes.

17.13 EXERCISES

17.1 Of the four OpenMP functions presented in this chapter, which two have the closest analogs to MPI functions? Name the MPI function each of these functions is similar to.

17.2 For each of the following code segments, use OpenMP pragmas to make the loop parallel, or explain why the code segment is not suitable for parallel execution.

a.
```
for (i = 0; i < (int) sqrt(x); i++) {
   a[i] = 2.3 * i;
   if (i < 10) b[i] = a[i];
}
```

b.
```
flag = 0;
for (i = 0; (i < n) & (!flag); i++) {
   a[i] = 2.3 * i;
   if (a[i] < b[i]) flag = 1;
}
```

c.
```
for (i = 0; i < n; i++)
   a[i] = foo(i);
```

d.
```
for (i = 0; i < n; i++) {
   a[i] = foo(i);
   if (a[i] < b[i]) a[i] = b[i];
}
```

e.
```
for (i = 0; i < n; i++) {
   a[i] = foo(i);
   if (a[i] < b[i]) break;
}
```

f.
```
dotp = 0;
for (i = 0; i < n; i++)
   dotp += a[i] * b[i];
```

g.
```
for (i = k; i < 2*k; i++)
   a[i] = a[i] + a[i-k];
```

h.
```
for (i = k; i < n; i++)
   a[i] = b * a[i-k];
```

17.3 Suppose OpenMP did not have the `reduction` clause. Show how to implement an efficient parallel reduction by adding a private variable and using the `critical` pragma. Illustrate your methodology using the π-estimation program segment from Section 17.5.

17.4 Section 17.7.3 discusses an interleaved scheduling of tasks to balance workloads among threads initializing an upper triangular matrix. Explain why increasing the chunk size from 1 could improve the cache hit rate.

17.5 Give an example of a simple parallel `for` loop that would probably execute faster with an interleaved static scheduling than by giving each task a single contiguous chunk of iterations. Your example should not have nested loops.

17.6 Give an original example of a parallel `for` loop that would probably execute in less time if it were dynamically scheduled rather than statically scheduled.

17.7 In Section 17.8 we develop a parallel code segment allowing multiple threads to work through a single "to-do" list. Explain how two threads could end up processing the same task if function `get_next_task` is not executed atomically.

17.8 Figure 17.9 illustrates a C/OpenMP program that uses the Monte Carlo algorithm to compute π. Note that the iterations of the `for` loop are divided among the threads explicitly. Implement another version of this program that uses the `for` pragma to delegate the allocation of loop iterations to the run-time system. Benchmark your program for various values of n (number of samples) and t (number of threads).

17.9 Use OpenMP directives to express as much parallelism as possible in the following code segment from Winograd's matrix multiplication algorithm (adapted from Baase and Van Gelder [5]).

```
for (i = 0; i < m; i++) {
   rowterm[i] = 0.0;
   for (j = 0; j < p; j++)
      rowterm[i] += a[i][2*j] * a[i][2*j+1];
}
for (i = 0; i < q; i++) {
   colterm[i] = 0.0;
   for (j = 0; j < p; j++)
      colterm[i] += b[2*j][i] * b[2*j+1][i];
}
```

17.10 Use OpenMP directives to implement a parallel program for the Sieve of Eratosthenes. Benchmark your program for various values of n and t (number of threads).

17.11 Use OpenMP directives to implement a parallel program for Floyd's algorithm (Figure 6.2). Benchmark your program for different values of n and t (number of threads).

17.12 Use OpenMP directives to implement a parallel version of matrix-vector multiplication (Figure 8.1). Benchmark your program for different values of n and t (number of threads).

17.13 Use OpenMP directives to implement a parallel program that solves a dense system of linear equations using Gaussian elimination with row pivoting, followed by back substitution (Figure 12.7). Benchmark your program for different values of n and t (number of threads).

17.14 Use OpenMP directives to implement a parallel version of the conjugate gradient method (Figure 12.13), assuming the coefficient matrix A is symmetrically banded. Benchmark your program for different values of n and t (number of threads).

17.15 Use OpenMP directives to implement a parallel version of Parallel Sorting by Regular Sampling (Figure 14.5). Benchmark your program for different values of n and t (number of threads).

17.16 Use OpenMP directives to implement a parallel program that solves the 15-puzzle (Chapter 17). For a variety of scrambled puzzles, benchmark your program for different values of t (number of threads).

CHAPTER

18

Combining MPI and OpenMP

The good things in life are not to be had singly, but come to us with a mixture.
Charles Lamb, *That You Must Love Me and Love My Dog*

18.1 INTRODUCTION

Most commercial multicomputers with hundreds or thousands of CPUs are actually collections of centralized multiprocessors, and many commodity clusters are made up of dual-processor or even quad-processor nodes. For these reasons it is good to know how to transform a program using MPI into a program using both MPI and OpenMP, suitable for execution on a multicomputer cluster of multiprocessors.

It is true that you can execute an MPI-only C program on a multiprocessor cluster by creating one MPI process for each CPU on the system (Figure 18.1a). In this case some MPI processes will happen to be on the same multiprocessor, but all process interactions will happen via message-passing. Sometimes, however, it is better to construct a hybrid parallel program (Figure 18.1b). In this case one MPI process executes on each multiprocessor. Inside parallel sections of code the MPI processes fork threads to occupy the multiprocessor CPUs, and these threads can interact via shared variables.

In many cases hybrid programs using both MPI and OpenMP execute faster than programs using only MPI.

Sometimes hybrid programs execute faster because they have lower communication overhead. Suppose we are executing our program on a cluster of m multiprocessors, where each multiprocessor has k CPUs. In order to utilize every CPU, a program relying on MPI must create mk processes. During communication steps, mk processes are active. On the other hand, a hybrid program need only create m processes. In parallel sections of code, the workload is divided among k threads on each multiprocessor. Hence every CPU is utilized. However, during

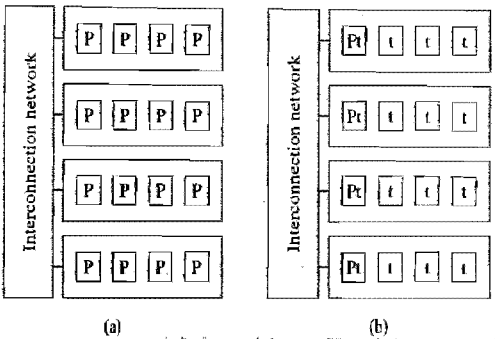


Figure 18.1 Two ways to execute parallel programs on multiprocessor clusters. (a) Create an MPI process (P) for every CPU. (b) Create an MPI process (P) for every multiprocessor and create threads (t) to occupy the CPUs.

communication steps, only m processes are active. This may well give the hybrid program lower communication overhead than a "pure" MPI program, resulting in higher speedup.

Another way a hybrid program can achieve higher speedup is if it is practical to parallelize some portions of the computation via lighter-weight threads, but not via heavier-weight processes. Consider the following example.

Suppose we are parallelizing a serial program that executes in 100 seconds. The program spends five seconds (5 percent of the execution time) performing inherently sequential operations. It spends 90 seconds (90 percent of the execution time) performing operations that are perfectly parallelizable. We translate this portion of the program into parallel code that achieves linear speedup.

The last five seconds (5 percent) are spent doing operations that could be performed in parallel, but require substantial communication overhead. The time required for the MPI function calls is so great that it is not worth making these operations parallel. Instead, we will replicate these operations on the MPI processes.

However, suppose that in a shared-memory environment it is practical to execute these operations in parallel. Suppose further that if these operations are performed on two processors, the parallel overhead is negligible.

Let's compute the speedup our program will achieve, assuming we are executing it on a cluster of eight dual-processor systems. With 16 MPI processes, the maximum speedup achievable (by Amdahl's Law) is

$$\frac{1}{(0.10 + 0.90/16)} = 6.4$$

Alternately, we can execute the program on eight MPI processes and allow double-threaded execution within each process. For the 90 percent of the code that is perfectly parallelizable, the 16 threads execute these operations 16 times faster. For the 5 percent of the code that is replicated across the nodes, two threads

will execute it twice as fast. The 5 percent of the code that is inherently sequential remains. The maximum speedup achievable is

$$\frac{1}{0.05 + 0.05/2 + 0.90/16} = 7.6$$

In this case the hybrid program is 19 percent faster than the program using only MPI.

A third situation in which hybrid parallelism can be useful is when some MPI processes are idle while others are busy. Suppose an application is executing on a multiprocessor cluster. On one of the multiprocessors three of the MPI processes are waiting for messages, while the fourth process is active. If the active process could exploit the idle CPUs to execute some parallel operations faster, then it would be worthwhile to fork some threads.

In this chapter we'll illustrate the transformation of programs using MPI into programs using both MPI and OpenMP through two case studies. The first case study is an implementation of the conjugate gradient algorithm described in Chapter 12. The second case study is an implementation of the Jacobi method described in Chapter 13. In both cases relatively small changes to the MPI-only C program are sufficient to translate it into an MPI/OpenMP C program that achieves significantly higher speedup.

18.2 CONJUGATE GRADIENT METHOD

18.2.1 MPI Program

The program presented in Figure 18.2 implements the conjugate gradient method to solve a system of linear equations $Ax = b$, where the coefficient matrix A is positive definite. The program design is based upon a block-row decomposition of matrix A among the processes. It assumes vector b and all other vectors are replicated.

Function `main` invokes the usual sequence of MPI start-up functions, then uses two of the utility functions developed in earlier chapters to read matrix A and vector b from files. If matrix A is not square, or if the number of columns in A does not match the number of rows in b, the algorithm terminates. Assuming the matrix is $n \times n$ and the vector has n elements, function `main` allocates space for the solution vector x, calls function `cg` to solve $Ax = b$ for x, and prints the solution.

Function `cg` is a straightforward implementation of the conjugate gradient method as described in Chapter 12. Code to initialize, add, and subtract vectors appears in line, but dot (inner) products and matrix-vector multiplications are accomplished through function calls. The conjugate gradient method is an iterative algorithm that generates successively better approximations to the solution vector. The function terminates either when it has converged on the solution or when it has

```
/* Conjugate Gradient Method in MPI */

#include <stdlib.h>
#include <stdio.h>
#include "mpi.h"
#include "MyMPI.h"

main (int argc, char *argv[])
{
   double **a;                /* Solving Ax = b for x */
   double *astorage;          /* Holds elements of A */
   double *b;                 /* Constant vector */
   double *x;                 /* Solution vector */
   int     p;                 /* MPI Processes */
   int     id;                /* Process rank */
   int     m;                 /* Rows in A */
   int     n;                 /* Columns in A */
   int     nl;                /* Elements in b */

   MPI_Init (&argc, &argv);
   MPI_Comm_size (MPI_COMM_WORLD, &p);
   MPI_Comm_rank (MPI_COMM_WORLD, &id);
   read_block_row_matrix (id, p, argv[1], (void *) &a,
      (void *) &astorage, MPI_DOUBLE, &m, &n);
   nl = read_replicated_vector (id, p, argv[2],
         (void **) &b, MPI_DOUBLE);
   if ((m != n) || (n != nl)) {
      if (!id)
         printf ("Incompatible dimensions (%dx%d) x (%d)\n",
            m, n, nl);
   } else {
      x = (double *) malloc (n * sizeof(double));
      cg (p, id, a, b, x, n);
      print_replicated_vector (id, p, x, MPI_DOUBLE, n);
   }
   MPI_Finalize();
}
```

Figure 18.2 MPI program implementing the conjugate gradient method.

iterated n times, whichever comes first. The algorithm typically finds a solution in far fewer than n iterations.

Function `dot_product`, when passed two vectors, returns a double-precision scalar value that is the dot product of the two vectors. Since all vectors are replicated, every process has all the values it needs to compute any dot product—no communications are needed. The function has time complexity $\Theta(n)$.

Function `matrix_vector_product`, when passed a matrix and a vector, returns through another parameter the matrix-vector product, another vector. Each process is assigned a contiguous group of rows of the matrix, while vectors are replicated. Hence multiplying these rows times a vector results in the solution being distributed in blocks across the set of processes. The computational complexity of this function is $\Theta(n^2/p)$.

```
#define EPSILON  1.0e-10  /* Convergence criterion */
double *piece;            /* Temp storage allocated once */

/* Conjugate gradient method solves ax = b for x */

cg (int p, int id, double **a, double *b, double *x, int n )
{
   int     i, it;                 /* Loop indices */
   double *d;
   double *g;                     /* Gradient vector */
   double  denom1, denom2, num1,
           num2, s, *tmpvec;      /* Temporaries */

   double dot_product (double , double , int);
   void matrix_vector_product (int, int, int, double ,
                               double *, double *);

   /* Initialize gradient vectors */

   d = (double *) malloc (n * sizeof(double));
   g = (double *) malloc (n * sizeof(double));
   tmpvec = (double *) malloc (n * sizeof(double));
   piece = (double *) malloc (BLOCK_SIZE(id,p,n) *
                              sizeof(double));
   for (i = 0; i < n; i++) {
      d[i] = 0.0;
      x[i] = 0.0;
      g[i] = -b[i];
   }

   /* Algorithm converges in n or fewer iterations */

   for (it = 0; it < n; it++) {
      denom1 = dot_product (g, g, n);
      matrix_vector_product (id, p, n, a, x, g);
         for (i = 0; i < n; i++)
         g[i] -= b[i];
      num1 = dot_product (g, g, n);

      /* When g is sufficiently close to 0, time to halt */

      if (num1 < EPSILON) break;

      for (i = 0; i < n; i++)
         d[i] = -g[i] + (num1/denom1) * d[i];
      num2 = dot_product (d, g, n);
      matrix_vector_product (id, p, n, a, d, tmpvec);
      denom2 = dot_product (d, tmpvec, n);
      s = -num2 / denom2;
      for (i = 0; i < n; i++) x[i] += s * d[i];
   }
}
```

Figure 18.1 (contd.) MPI program implementing the conjugate gradient method.

```
/*
 *   Return the dot product of two vectors
 */

double dot_product (double *a, double *b, int n)
{
   int i;
   double answer;

   answer = 0.0;
   for (i = 0; i < n; i++)
      answer += a[i] * b[i];
   return answer;
}

/*
 *   Compute the product of matrix a and vector b and
 *   store the result in vector c.
 */

void matrix_vector_product (int id, int p, int n,
                            double **a, double *b, double *c)
{
   int    i, j;
   double tmp;       /* Accumulates sum */

   for (i = 0; i < BLOCK_SIZE(id,p,n); i++) {
      tmp = 0.0;
      for (j = 0; j < n; j++)
         tmp += a[i][j] * b[j];
      piece[i] = tmp;
   }
   new_replicate_block_vector (id, p, piece, n,
      (void *) c, MPI_DOUBLE);
}
```

Figure 18.1 (contd.) MPI program implementing the conjugate gradient method.

In order to replicate the solution vector, function `matrix_vector_product` calls function `new_replicate_block_vector`. Function `new_replicate_block_vector` differs from function `replicate_block_vector` in that it does not allocate space for the replicated vector. Instead, the calling function passes a pointer to the memory where the replicated vector should be stored. Using the new function saves time, because it saves function `matrix_vector_product` from having to copy the replicated vector to where it is needed. Replicating a block vector is an example of an all-gather operation that has time complexity $\Theta(\log p + n)$.

We see that the most computationally intensive portion of the conjugate gradient method is in the matrix-vector multiplications. In addition, the only communications required inside the method's `while` loop occur as part of the matrix-vector multiplications.

Table 18.1 Result of profiling the conjugate gradient program on one and eight CPUs of a commodity cluster.

Function	1 CPU	8 CPUs
`matrix_vector_product`	99.55%	97.49%
`dot_product`	0.19	1.06
`cg`	0.25	1.44

18.2.2 Functional Profiling

Our first step is to profile the program to discover where the greatest opportunities for further parallelization lie.

We insert calls to function `MPI_Wtime` inside the program to monitor the amount of time spent inside each of the principal functions (`cg`, `dot_product`, and `matrix_vector_product`). We record the average time spent inside each function as the program solves systems of size 768 on 1 CPU and on 8 CPUs of a commodity cluster. Table 18.1 reveals the results of this benchmarking. (The figure for function `cg` excludes time spent in the functions it calls.)

As you can see, virtually all of the execution time is spent within function `matrix_vector_product`. This makes sense, since it is the part of the algorithm with the highest computational complexity. This function should be the focus of our parallelization.

18.2.3 Parallelizing Function `matrix_vector_product`

Function `matrix_vector_product` has nested `for` loops. We maximize grain size by parallelizing the outermost possible loop. The outer loop, indexed by `i`, computes element `i` of the result vector. While various matrix and vector values are read inside the loop, the only values written are `tmp`, `j`, and `piece[i]`. Every iteration of the outer loop may be executed in parallel if each thread has a private copy of `tmp` and `j`.

We can use the `parallel for` pragma to make parallel the loop indexed by `i`. The `for` loop index `i` is private by default. As we noted earlier, each thread needs its own copy of `tmp` and `j`, so we declare them to be private variables. Our completed pragma is

```
#pragma omp parallel for private(j,tmp)
```

We also want to give the user the opportunity to specify the number of active threads per process. To do this, we add a call to `omp_set_num_threads` to function `main`. Its argument comes from the command line. We put this function call into `main` immediately after the call to `MPI_Comm_rank`:

```
omp_set_num_threads(atoi(argv[3]));
```

We have added only two lines to the program using MPI to transform it into a program using both MPI and OpenMP!

18.2.4 Benchmarking

Now let's benchmark our program on a commodity cluster containing four dual-processor computers. First we run the original program. When it executes on one, two, three, and four processes (one to four CPUs), each process is on a different node. By doing this we maximize memory bandwidth to the CPUs. When we execute five processes on five CPUs, two of the processes are on the same node. By the time we get to eight processes, a process is assigned to every available CPU, two per computer.

In our MPI/OpenMP benchmarking, we create only one MPI process per computer. In our first experiment we run one process with two threads on one computer. Our second experiment runs two processes with a total of four threads on two computers, and so on. Our last experiment runs four processes and eight threads, so that one thread is assigned to every available CPU. In other words, we execute the program using MPI and OpenMP on two, four, six, and eight CPUs.

Figure 18.2 illustrates the results of our benchmarking. Each time plotted in the graph represents the average of five executions of the program. The original program executes faster than the hybrid program when two CPUs are used. This makes sense, because both OpenMP threads are executing on the same computer. The CPUs executing the OpenMP threads will have lower memory bandwidth and a lower cache hit rate. By the same reasoning, we'd expect the program using only MPI to execute faster when four CPUs are used, and this is the case. However,

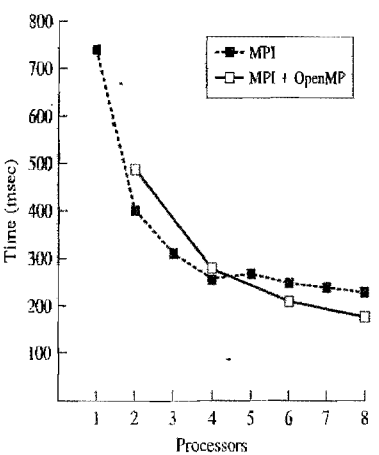


Figure 18.2 Result of benchmarking the original and hybrid parallel programs using the conjugate gradient method to solve a dense system of 768 equations. All times are in milliseconds. The target architecture is a commodity cluster containing four dual-processor nodes.

once we get to six CPUs, computers are being shared for pairs of MPI processes, so the previous advantage disappears. At this point the lower communication cost of the hybrid program begins to pay off. The final data point shows that executing four MPI processes, each with two threads, results in 27 percent lower execution time than executing eight MPI processes.

18.3 JACOBI METHOD

18.3.1 Profiling MPI Program

For our second case study we will look at a more complicated example. We have written a C/MPI program that uses the Jacobi method to solve the steady-state heat distribution problem, as described in Chapter 13. To make our example easier to understand, we have chosen a rowwise block-striped decomposition of the two-dimensional matrix representing the finite difference mesh. The program's execution is divided into three phases. Function `initialize_mesh` is responsible for allocating a process's portion of the matrix and initializing both the boundary and interior values. Function `find_steady_state` implements the Jacobi method for solving the partial differential equation. It iterates until the values at the mesh points have converged. Function `print_solution` prints to standard output the values at each mesh point.

Our first step is to profile the parallel program's execution on one and four processors of a commodity cluster. The results are summarized in Table 18.2. The vast majority of time is spent inside function `find_steady_state`. For that reason we will focus our parallelization efforts on this function, which appears in Figure 18.3.

18.3.2 Parallelizing Function `find_steady_state`

Except for two early initializations and a return statement, function `find_steady_state` consists of a `for` loop. There are many reasons we cannot execute the loop in parallel. It is not in canonical form. It contains a `break` statement. It contains calls to MPI functions. The most significant reason, however, is that there are data dependences between iterations. Each iteration relies on values computed in the previous iteration. So we need to look for parallelism inside an iteration of the outer `for` loop.

Table 18.2 Result of profiling a C/MPI program implementing the Jacobi method. The target architecture is a commodity cluster.

Function	1 CPU	4 CPUs
`initialize_mesh`	0.01%	0.03%
`find_steady_state`	98.48%	93.49%
`print_solution`	1.51%	6.48%

```
int find_steady_state (int p, int id, int my_rows,
                       double **u, double **w)
{
   double diff;
   double global_diff;
   int its;
   MPI_Status status;
   int i, j;

   its = 0;
   for (;;) {
      if (id > 0)
         MPI_Send (u[1], N, MPI_DOUBLE, id-1, 0, MPI_COMM_WORLD);
      if (id < p-1) {
         MPI_Send (u[my_rows-2], N, MPI_DOUBLE, id+1, 0, MPI_COMM_WORLD);
         MPI_Recv (u[my_rows-1], N, MPI_DOUBLE, id+1, 0, MPI_COMM_WORLD,
            &status);
      }
      if (id > 0)
         MPI_Recv (u[0], N, MPI_DOUBLE, id-1, 0, MPI_COMM_WORLD, &status);
      diff = 0.0;
      for (i = 1; i < my_rows-1; i++)
         for (j = 1; j < N-1; j++) {
            w[i][j] = (u[i-1][j] + u[i+1][j] +
                       u[i][j-1] + u[i][j+1])/4.0;
            if (fabs(w[i][j] - u[i][j]) > diff)
               diff = fabs(w[i][j] - u[i][j]);
         }
      for (i = 1; i < my_rows-1; i++)
         for (j = 1; j < N-1; j++)
            u[i][j] = w[i][j];
      MPI_Allreduce (&diff, &global_diff, 1, MPI_DOUBLE, MPI_MAX,
         MPI_COMM_WORLD);
      if (global_diff <= EPSILON) break;
      its++;
   }
}
```

Figure 18.3 More than 90 percent of the program's execution time is spent inside function `find_steady_state`. This is where we will look for opportunities for parallelization through multithreading.

Our focus shifts to the first `for` loop indexed by `i`. This is the loop that iterates through the rows of this process's share of the matrix, computing elements of `w` from elements of `u`. These assignment statements are independent of each other and may execute simultaneously.

However, when the absolute value of `w[i][j]` is greater than the current value of `diff`, we need to update `diff`. If we want multiple threads to reference the shared variable `diff`, we would need to put the `if` statement inside a critical section. This would reduce speedup.

Instead, we are going to introduce a new, private variable called `tdiff`. Each thread will initialize its copy of `tdiff` to zero before the `for` loop indexed by `i` and compare each value of `w[i][j]` it computes with `tdiff`. Since all the

threads are assigning values to different elements of w and different private copies of `tdiff`, we can use the `for` pragma to indicate that the `for` loop indexed by `i` may be made parallel.

The second `for` loop indexed by `i` copies elements of `w` to the corresponding elements of `u`. We place a `for` pragma before this loop to instruct the compiler to make it parallel.

After the second `for` loop indexed by `i` we create a critical section in which each thread compares its value of `tdiff` with the value of shared variable `diff`, and updates the value of `diff` to `tdiff` when `tdiff` is larger.

Note that creating private variable `tdiff` allows us to build a solution in which each thread only enters a critical section once per iteration of the Jacobi method. If all threads had referenced `diff` in the original pair of nested loops, each thread would have entered a critical section `i` times `j` times per iteration of the Jacobi method.

We use curly braces to create a block of code surrounding the two `for` pragmas, the `critical` pragma, and the statement that initializes private variable `tdiff`. At the top of this block of code we insert a `parallel` pragma. Our modified version of function `find_steady_state` appears in Figure 18.4.

We also modify function `main`, adding the statement

```
omp_set_num_threads (atoi(argv[1]));
```

so that the user can specify the number of active threads per MPI process from the command line.

18.3.3 Benchmarking

We benchmark both the original program and the hybrid program on a commodity cluster containing four dual-processor nodes. First we run the C program using only MPI. Our allocation of processes to nodes is the same as in the previous example. When executing on one to four CPUs, each process is on a different node, to maximize memory bandwidth to the CPUs. When we execute five processes on five CPUs, two of the processes are on the same node. By the time we get to eight processes, a process is assigned to every available CPU, two per computer.

In our MPI/OpenMP benchmarking, we create only one MPI process per node. Two threads are associated with each process. Hence our four data points represent two, four, six, and eight active CPUs.

Figure 18.5 illustrates the results of our benchmarking. Each time plotted in the graph represents the average of five executions of the program. Note that the hybrid program is uniformly faster than the original program executing on the same number of CPUs. This is because the computation/communication ratio of the hybrid program is superior. The number of mesh points updated per element communicated is twice as high per node for the hybrid program. The lower communication overhead leads to a higher speedup. On eight CPUs the hybrid program is 19 percent faster than the parallel program relying solely on MPI.

```
int find_steady_state (int p, int id, int my_rows,
                       double **u, double **w)
{
   double     diff;
   double     global_diff;
   int        i, j;
   int        its;
   double     tdiff;
   MPI_Status status;

   its = 0;
   for (;;) {
      if (id > 0)
         MPI_Send (u[1], N, MPI_DOUBLE, id-1, 0, MPI_COMM_WORLD);
      if (id < p-1) {
         MPI_Send (u[my_rows-2], N, MPI_DOUBLE, id+1, 0, MPI_COMM_WORLD);
         MPI_Recv (u[my_rows-1], N, MPI_DOUBLE, id+1, 0, MPI_COMM_WORLD,
            &status);
      }
      if (id > 0)
         MPI_Recv (u[0], N, MPI_DOUBLE, id-1, 0, MPI_COMM_WORLD, &status);
      diff = 0.0;
#pragma omp parallel private (i, j, tdiff)
{
      tdiff = 0.0;
#pragma omp for
      for (i = 1; i < my_rows-1; i++)
         for (j = 1; j < N-1; j++) {
            w[i][j] = (u[i-1][j] + u[i+1][j] +
                       u[i][j-1] + u[i][j+1])/4.0;
            if (fabs(w[i][j] - u[i][j]) > tdiff)
               tdiff = fabs(w[i][j] - u[i][j]);
         }
#pragma omp for nowait
      for (i = 1; i < my_rows-1; i++)
         for (j = 1; j < N-1; j++)
            u[i][j] = w[i][j];
#pragma omp critical
      if (tdiff > diff) diff = tdiff;
}
      MPI_Allreduce (&diff, &global_diff, 1, MPI_DOUBLE, MPI_MAX,
         MPI_COMM_WORLD);
      if (global_diff <= EPSILON) break;
      its++;
   }
   return its;
}
```

Figure 18.4 Function `find_steady_state` after OpenMP pragmas have been inserted.

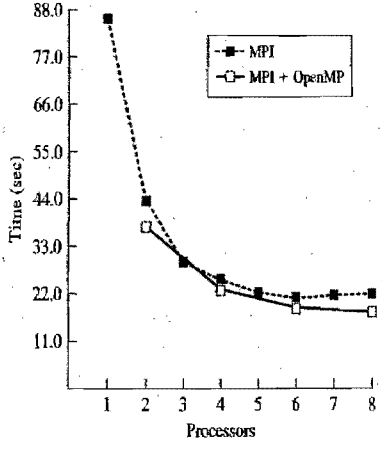


Figure 18.5 Result of benchmarking the original and hybrid parallel programs using the Jacobi method to solve the steady-state heat equation on a 200 x 200 grid. All times are in seconds. The target architecture is a commodity cluster containing four dual-processor nodes.

18.4 SUMMARY

Many contemporary parallel computers, including most of the world's fastest systems, consist of a collection of multiprocessors. While it is possible to program a collection of multiprocessors solely using MPI, you can often improve performance by using both MPI and OpenMP. MPI handles the larger-grained communications among multiprocessors, while the lighter-weight threads of OpenMP handle the processor interactions within each multiprocessor.

In this chapter we have looked at two examples of transforming a C program with MPI calls program into a hybrid program suitable for execution on a cluster of multiprocessors. The first step of the transformation is profiling the original parallel program to discover the functions consuming the most CPU cycles. These need to be the focus of the parallelization effort. Often, relatively few function calls and/or pragmas need to be added to the original program to complete its conversion into a mixed code that leverages the strengths of both MPI and OpenMP.

18.5 EXERCISES

18.1 Which functions of the document classification program of Chapter 9 are most likely to be suitable for parallelization with OpenMP pragmas?

(Consider all of the program's functions, not simply those listed in Figure 9.7.) Justify your answer.

18.2 Suppose your institution's computer center operates a commodity cluster made out of multiprocessor nodes. Suppose further that most of the cluster's CPU cycles are devoted to executing Monte Carlo methods written in programs using the MPI library. The director of the computer center would like these programs to execute faster. Knowing what you do about Monte Carlo methods, what are the prospects for significantly improving their performance by converting them into hybrid programs that also invoke OpenMP pragmas? Defend your position.

18.3 Convert the program implementing Floyd's algorithm (Figure 6.9) into a hybrid program that includes OpenMP pragmas. Set the number of threads equal to the number of processors available to the program. Contrast the speedup achieved by your program with the original program.

18.4 Convert the matrix-vector multiplication program appearing in Figure 8.8 into a hybrid program that includes OpenMP pragmas. Set the number of threads equal to the number of processors available to the program. Contrast the speedup achieved by your program with the original program.

18.5 Convert the matrix-vector multiplication program appearing in Figure 8.14 into a hybrid program that includes OpenMP pragmas. Set the number of threads equal to the number of processors available to the program. Contrast the speedup achieved by your program with the original program.

18.6 Write a hybrid C program with both MPI function calls and OpenMP pragmas that solves a dense system of linear equations using Gaussian elimination followed by back substitution. Benchmark your program for various values of n and p, where p is the number of multiprocessor nodes being used.

18.7 Write a hybrid program implementing Parallel Sorting by Regular Sampling. Benchmark your program for various values of n and p, where p is the number of multiprocessor nodes being used.

18.8 Write a hybrid program implementing the fast Fourier transform. Benchmark your program for various values of n and p, where p is the number of multiprocessor nodes being used.

18.9 Write a hybrid program solving the n-queens problem. Benchmark your program for various values of n and p, where p is the number of multiprocessor nodes being used.

18.10 Write a hybrid program to solve the 15-puzzle. Benchmark your program for various values of n and p, where p is the number of multiprocessor nodes being used.

MPI Functions

This appendix describes every function in the MPI-1 standard. Every parameter is commented with one of these three notations:

- IN (input parameter)—the caller provides the value
- OUT (output parameter)—the function sets the value
- IN/OUT (input/output parameter)—the value is set by both the caller and the function

```
int MPI_Abort (
   MPI_Comm comm,    /* IN - Communicator */
   int error_code    /* IN - Error code */
)
```

`MPI_Abort` makes a "best effort" attempt to abort all processes in the specified communicator. It returns the error code to the calling environment.

```
int MPI_Address (
   void *location,     /* IN - A location in 'offsets' */
   MPI_Aint *offsets   /* IN - Array of addresses */
)
```

Function `MPI_Address` returns the byte address of `location` in array `offsets`. It is useful when building derived datatypes.

```
int MPI_Allgather (
   void *send_buffer,   /* IN - Send buffer */
   int send_cnt,        /* IN - Elements in send buffer */
   MPI_Datatype
      send_dtype,       /* IN - Send buffer element type */
   void *recv_buffer,   /* OUT - Receive buffer */
   int recv_cnt,        /* IN - Elements gathered
                                from each process */
```

```
    MPI_Datatype
       recv_dtype,      /* IN - Receive buffer element type */
    MPI_Comm comm       /* IN - Communicator */
)
```

`MPI_Allgather` is a collective communication function that performs an all-gather operation. All processes gather `send_cnt` elements from every process in the communicator. When the function returns, the concatenation of these elements is in `recv_buffer` of every process. Use `MPI_Allgatherv` if different processes contribute different numbers of elements to the gather or if the elements are not concatenated in process rank order.

```
int MPI_Allgatherv (
    void *send_buffer, /* IN - Send buffer */
    int send_cnt,      /* IN - Number of elements sent by
                             this process */
    MPI_Datatype
       send_dtype,     /* IN - Send buffer element type */
    void *recv_buffer, /* OUT - Receive buffer */
    int *recv_cnts,    /* IN - Group-sized array. Entry j is
                             number of elements to receive
                             from process j */
    int *recv_disp,    /* IN - Group-sized array. Entry j is
                             the offset from the start of
                             recv_buffer where the elements
                             received from process j should
                             be put */
    MPI_Datatype
       recv_dtype,     /* IN - Receive buffer element type */
    MPI_Comm comm      /* IN - Communicator */
)
```

`MPI_Allgatherv` is a collective communication function that performs an all-gather operation. The number of elements contributed by each process may vary. Array `disp` indicates where in `recv_buffer` each process's chunk should be placed; the pieces need not be assembled in process rank order. When the function returns, the gathered elements are in `recv_buffer` of every process. Use the simpler `MPI_Allgather` if all processes contribute the same numbers of elements to the gather and the elements are concatenated in process rank order.

```
int MPI_Allreduce (
    void *send_buffer, /* IN - Send buffer */
    void *recv_buffer, /* OUT - Receive buffer. */
    int cnt,           /* IN - Number of elements to reduce */
    MPI_Datatype dtype,/* IN - Element type */
    MPI_Op op,         /* IN - Reduction operator */
    MPI_Comm comm      /* IN - Communicator */
)
```

MPI_Allreduce is a collective communication function that performs cnt reductions. When the function returns, all processes have the results of the reductions. Use MPI_Reduce if only a single process needs the results of the reductions.

```
int MPI_Alltoall (
   void *send_buffer,          /* IN - Send buffer */
   int send_cnt,               /* IN - Elements sent to each
                                      process */
   MPI_Datatype send_dtype,    /* IN - Sent element type */
   void *recv_buffer,          /* OUT - Receive buffer */
   int recv_cnt,               /* IN - Elements received from
                                      each process */
   MPI_Datatype recv_dtype,    /* IN - Received element type */
   MPI_Comm comm               /* IN - Communicator */
)
```

MPI_Alltoall performs an all-to-all exchange within a communicator. Each process sends (and receives) the same number of elements to (and from) every process, including itself. Use the more general function MPI_Alltoallv if the number of elements sent from any process to any other process is not a constant.

```
int MPI_Alltoallv (
   void *send_buffer, /* IN - Send buffer */
   int *send_cnts,    /* IN - Group-sized array. Entry j
                            indicates number of elements of
                            send_buffer to send to
                            process j */
   int *send_disp,    /* IN - Group-sized array. Entry j
                            indicates the displacement from
                            the start of send_buffer of the
                            elements sent to process j */
   MPI_Datatype
         send_dtype, /* IN - Send buffer element type */
   void *recv_buffer, /* OUT - Receive buffer */
   int *recv_cnts,    /* IN - Group-sized array. Entry j is
                            the number of elements being
                            received from process j. */
   int *recv_disp,    /* IN - Group-sized array. Entry j is
                            the displacement from the
                            start of recv_buffer where the
                            elements received from process
                            j should be stored. */
   MPI_Datatype
         recv_dtype, /* IN - Receive buffer element type */
   MPI_Comm comm      /* IN - Communicator */
)
```

Function MPI_Alltoallv performs an all-to-all data exchange. Use the simpler function MPI_Alltoall if each process contributes the same number of elements to the exchange and the received elements are concatenated in process rank order.

```
int MPI_Attr_delete (
   MPI_Comm comm,      /* IN - Communicator */
   int key             /* IN - Attribute identifier */
)
```

MPI_Attr_delete deletes the cached attribute corresponding to key.

```
int MPI_Attr_get (
   MPI_Comm comm,      /* IN - Communicator */
   int key,            /* IN - Attribute identifier */
   void *attr,         /* OUT - Pointer to attribute */
   int *flag           /* OUT - Existence flag */
)
```

Function MPI_Attr_get returns through attr a pointer to a previously cached attribute with identifier key. Successful retrieval is indicated by the return value of flag. It is 1 if the attribute was retrieved, and 0 otherwise.

```
int MPI_Attr_put (
   MPI_Comm comm,      /* IN - Communicator */
   int key,            /* IN - Attributed identifier */
   void *attr          /* IN - Pointer to attribute */
)
```

Function MPI_Attr_put associates the integer value of key with the attribute record pointed to by attr.

```
int MPI_Barrier (
   MPI_Comm            /* IN - Communicator */
)
```

MPI_Barrier is a collective communication function that performs a barrier synchronization among all processes in the specified communicator.

```
int MPI_Bcast (
   void *buffer,       /* IN/OUT - Message address */
   int cnt,            /* IN - Elements in message */
   MPI_Datatype dtype, /* IN - Element type */
   int root,           /* IN - Rank of root process */
   MPI_Comm comm       /* IN - Communicator */
)
```

Function MPI_Bcast is a collective communication operation allowing one process to broadcast a message to all other processes in a communicator. Parameter root is the rank of the process with the message to broadcast.

Passed the handle to a Cartesian communicator and the number of dimensions in the grid, `MPI_Cart_get` returns the size of each grid dimension, whether or not each dimension is periodic (wraps around), and the coordinates of the process calling the function.

```
int MPI_Cart_map (
    MPI_Comm comm,      /* IN - Cartesian communicator */
    int dims,           /* IN - Grid dimensions */
    int *size,          /* IN - Size of each grid dimension */
    int *periodic,      /* IN - Periodicity of each dimension */
    int *new_rank       /* OUT - "Optimized" process rank */
)
```

Passed the handle to a Cartesian communicator, the number of grid dimensions, the size of each dimension, and information about whether each dimension is periodic (wraps around), function `MPI_Cart_map` returns the "optimized" rank of the calling process through `new_rank`.

```
int MPI_Cart_rank (
    MPI_Comm comm,      /* IN - Cartesian communicator */
    int *coords,        /* IN - Process coordinates */
    int *rank           /* OUT - Process rank */
)
```

Passed a Cartesian communicator handle and the coordinates of a process, function `MPI_Cart_rank` returns the rank of that process.

```
int MPI_Cart_shift (
    MPI_Comm comm,      /* IN - Cartesian communicator */
    int shift_dim,      /* IN - Dimension of shift */
    int direction,      /* IN - >0 up; <0 down */
    int *src,           /* OUT - Source of received message */
    int *dest           /* OUT - Destination of sent message */
)
```

Function `MPI_Cart_shift` provides the calling process with the source and destination information it needs to perform a send-receive operation along a particular dimension of a Cartesian grid. If there is no wraparound, the function returns `MPI_PROC_NULL` in `src` and/or `dest` to indicate out-of-range shifts.

```
int MPI_Cart_sub (
    MPI_Comm comm,          /* IN - Cartesian communicator */
    int *free,              /* IN - Array of size dimensions.
                                    Entry free[i] is 1 if coord i
                                    can vary, 0 otherwise. */
    MPI_Comm *new_comm      /* OUT - Handle to new communicator */
)
```

Function MPI_Cart_sub partitions a Cartesian grid into multiple grids of lower dimension. The number of dimensions in the new grids is equal to the number of elements of free that have value 0. The function returns, through new_comm, a handle to the new communicator to which the calling process belongs.

```
int MPI_Cartdim_get (
   MPI_Comm comm,          /* IN - Cartesian communicator */
   int *dims               /* OUT - Dimensions */
)
```

Function MPI_Cartdim_get returns the number of dimensions in a Cartesian communicator.

```
int MPI_Comm_compare (
   MPI_Comm comm1,         /* IN - First communicator */
   MPI_Comm comm2,         /* IN - Second communicator */
   int *result             /* OUT - Result of comparison */
)
```

Function MPI_Comm_compare compares two communicators. The result of the comparison, returned through result, may be MPI_IDENT if the contexts and the groups are the same; MPI_CONGRUENT if the contexts are different but the groups contain the same processes with the same ranks; MPI_SIMILAR if the contexts and process ranks are different but the groups contain the same processes; and MPI_UNEQUAL otherwise.

```
int MPI_Comm_create (
   MPI_Comm old_comm,      /* IN - Old communicator */
   MPI_Group group,        /* IN - Process group */
   MPI_Comm *new_comm      /* OUT - New communicator */
)
```

The collective function MPI_Comm_create creates a new communicator from the processes listed in group.

```
int MPI_Comm_dup (
   MPI_Comm old_comm,      /* IN - Old communicator */
   MPI_Comm *new_comm      /* OUT - New communicator */
)
```

The collective function MPI_Comm_dup duplicates a communicator, returning a new communicator with the same group but a new context.

```
int MPI_Comm_free (
   MPI_Comm* comm          /* IN - Communicator */
)
```

The collective function MPI_Comm_free frees the resources associated with communicator comm.

```
int MPI_Comm_group (
   MPI_Comm comm,          /* IN - Communicator */
   MPI_Group *group        /* OUT - Process group */
)
```

Function MPI_Comm_group returns the process group associated with communicator comm.

```
int MPI_Comm_rank (
   MPI_Comm comm,  /* IN - Communicator */
   int *rank       /* OUT - Rank of calling process */
)
```

Function MPI_Comm_rank returns the rank of the calling process in a communicator.

```
int MPI_Comm_remote_group (
   MPI_Comm comm,     /* IN - Handle to inter-communicator */
   MPI_Group *procs   /* OUT - Handle to remote group */
)
```

Passed the handle to an inter-communicator, MPI_Comm_remote_group returns the remote process group.

```
int MPI_Comm_remote_size (
   MPI_Comm comm,     /* IN - Handle to inter-communicator */
   int *size          /* OUT - Remote group size */
)
```

Passed the handle to an inter-communicator, MPI_Comm_remote_size returns the number of processes in the remote group.

```
int MPI_Comm_size (
   MPI_Comm comm, /* IN - Communicator */
   int *size      /* OUT - Number of procs in communicator */
)
```

MPI_Comm_size returns the number of processes in a communicator.

```
int MPI_Comm_split (
   MPI_Comm old_comm,   /* IN - Old communicator */
   int partition,       /* IN - Partition number */
   int new_rank,        /* IN - Ranking value */
   MPI_Comm *new_comm   /* OUT - New communicator */
)
```

Collective function MPI_Comm_split partitions the processes in an existing communicator (old_comm) into one or more subgroups. Processes with the same value of partition are put in the same subgroup. Within a subgroup, processes are ranked according to the values of new_rank; ties are broken

according to the processes' ranks in old_comm. The function returns to each process a pointer to the new communicator to which it belongs.

```
int MPI_Comm_test_inter (
   MPI_Comm comm,        /* IN - Communicator */
   int *flag             /* OUT - Result of test */
)
```

Function MPI_Comm_test_inter, passed a communicator comm, sets flag to true if comm is an inter-communicator and false otherwise.

```
int MPI_Dims_create (
   int nodes,        /* IN - Number of grid nodes */
   int dims,         /* IN - Number of dimensions */
   int *size         /* OUT - Size of each dimension */
)
```

Passed the total number of nodes desired for a Cartesian grid and the number of grid dimensions, MPI_Dims_create returns an array of integers specifying the number of nodes in each dimension of the grid, so that the sizes of the dimensions are as balanced as possible.

```
int MPI_Errhandler_create (
   MPI_Handler_function
      *eh_func,          /* IN - Error handler function */
   MPI_Errhandler *eh    /* OUT - Handle to error handler */
)
```

Call function MPI_Errhandler_create to register function eh_func as an MPI exception handler. The function returns a pointer to the error handler, an opaque object.

User-created error handlers should be C functions of type MPI_Handler_function, which has this definition:

```
typedef void (MPI_Handler_function)(MPI_Comm *, int *, ...);
```

The first argument is the communicator in use. The second argument is the error code that should be returned by the MPI function raising the exception. The number and meaning of the remaining arguments are implementation dependent.

```
int MPI_Errhandler_free (
   MPI_Errhandler *eh      /* IN - Handle to error handler */
)
```

Function MPI_Errhandler_free marks for deallocation the error handler associated with eh, replacing it with MPI_ERRHANDLER_NULL.

```
int MPI_Errhandler_get (
   MPI_Comm comm,           /* IN - Communicator */
   MPI_Errhandler *eh_func  /* IN - Error handler function */
)
```

MPI_Errhandler_get associates, for the calling process, the error handler function eh_func with communicator comm.

```
int MPI_Errhandler_set (
   MPI_Comm comm,              /* IN - Communicator */
   MPI_Errhandler eh           /* IN - Error handler */
)
```

MPI_Errhandler_set associates, for the calling process, the error handler eh with communicator comm.

```
int MPI_Error_class (
   int code,                   /* IN - Error code */
   int *class                  /* OUT - Error class */
)
```

Error codes are implementation dependent. Error classes are part of the MPI standard. Passed an error code, function MPI_Error_class returns via the second parameter the error class that the code maps to.

```
int MPI_Error_string (
   int err_code,              /* IN - Error code */
   char *err_string,          /* OUT  - Error string */
   int *err_string_length     /* OUT - Length of error string */
)
```

Passed an error code or class, function MPI_error_string returns the error string associated with that code or class, as well as the length of the string. Allocate the buffer for err_string before calling the function; the buffer should be at least MPI_MAX_ERROR_STRING bytes long.

```
int MPI_Finalize (void)
```

MPI_Finalize terminates the MPI execution environment. Every process must call this function before exiting.

```
int MPI_Gather (
   void *send_buffer,      /* IN - Send buffer */
   int send_cnt,           /* IN - Elements in send buffer */
   MPI_Datatype
      send_dtype,          /* IN - Send buffer element type */
   void *recv_buffer,      /* OUT - Receive buffer */
   int recv_cnt,           /* IN - Number of elements gathered
                                  from each process */
   MPI_Datatype
      recv_dtype,          /* IN - Receive buffer element type */
   int root,               /* IN - Rank of gathering process */
   MPI_Comm comm           /* IN - Communicator */
)
```

`MPI_Gather` is a collective communication function that performs a gather operation. The root process gathers `send_cnt` elements from every process in the communicator (including itself). When the function returns, the result, a concatenation of elements, is in `recv_buffer`. Use `MPI_Gatherv` if different processes contribute different numbers of elements to the gather or if the elements are not concatenated in process rank order.

```
int MPI_Gatherv (
   void *send_buffer,  /* IN - Send buffer */
   int send_cnt,       /* IN - Elements in send buffer */
   MPI_Datatype
      send_dtype,      /* IN - Send buffer element type */
   void *recv_buffer,  /* OUT - Address of receive buffer */
   int *recv_cnt,      /* IN - Elements to gather from
                            each process */
   int *displacements, /* IN - Displacement in recv_buffer
                            of elements gathered from
                            each process */
   MPI_Datatype
      recv_dtype,      /* IN - Receive buffer element type */
   int root,           /* IN - Rank of gathering process */
   MPI_Comm comm       /* IN - Communicator */
)
```

`MPI_Gatherv` is a collective communication function that performs a gather operation. The root process gathers `send_cnt[i]` elements from every process `i` in the communicator (including itself). It puts the elements collected from process `i` in a contiguous group of elements of `recv_buffer` beginning with element `displacements[i]`. When the function returns, the result, a concatenation of elements, is in `recv_buffer`. Use the simpler function `MPI_Gather` if all processes contribute the same number of elements to the gather and the gathered elements are concatenated in process rank order.

```
int MPI_Get_count (
   MPI_Status *status, /* IN - Result of receive */
   MPI_Datatype dtype, /* IN - Type of elements received */
   int* cnt            /* OUT - Count of elements received */
)
```

Passed both a handle to the status variable containing the result of a receive operation and the type of the elements received, function `MPI_Get_count` returns the number of elements (*not bytes*) actually received.

```
int MPI_Get_elements (
   MPI_Status *status, /* IN - Result of receive */
   MPI_Datatype dtype, /* IN - Type of elements received */
   int* pe_cnt         /* OUT - Count of elements received */
)
```

Passed both a handle to the status variable associated with a receive operation and the type of the elements received, function MPI_Get_elements returns the number of primitive elements actually received.

```
int MPI_Get_processor_name (
   char *name,          /* OUT - Processor name */
   int *length          /* OUT - Length of processor name */
)
```

Function MPI_Get_processor_name returns the name of the physical processor on which the calling process is executing. Allocate buffer name before calling the function; the buffer should have length MPI_MAX_PROCESSOR_NAME.

```
int MPI_Get_version (
   int *major,      /* OUT - Major version number (1 or 2) */
   int *minor       /* OUT - Minor version number */
)
```

Function MPI_Get_version returns the major and minor MPI version numbers.

```
int MPI_Graph_create (
   MPI_Comm
      old_comm,    /* IN - Old communicator */
   int n,          /* IN - Nodes in process graph */
   int *degree,    /* IN - Array of size n with vertex degree
                           stored indirectly. Entry 0 is
                           degree of vertex 0. For all other
                           vertices i, degree[i]-degree[i-1]
                           is degree of vertex i. */
   int *edge,      /* IN - Array with rest of edge info. Entry
                           i is destination of edge i. */
   int reorder,    /* IN - Ranks changeable (logical) */
   MPI_Comm
      *graph_comm /* OUT - Graph communicator */
)
```

Function MPI_Graph_create returns a pointer to a new communicator containing information about the directed graph structure of a group of processes. If reorder = false, the system may not change the rank of the processes. Otherwise, it may reorder process ranks to improve efficiency. The source of each directed edge can be determined from array degree; the destination of each edge is stored in array edges.

```
int MPI_Graph_get (
   MPI_Comm comm,     /* IN - Graph communicator */
   int n,             /* IN - Number of vertices in graph */
   int m,             /* IN - Number of edges in graph */
```

```
    int *index,         /* OUT - Index information */
    int *edge           /* OUT - Edge information */
)
```

Passed a communicator associated with a graph topology, the number of vertices (processes) in the communicator, and the number of edges (connections between processes) in the communicator, `MPI_Graph_get` returns arrays `index` and `edge` that together represent the structure of the graph. See the description of function `MPI_Graph_create` for an explanation of the graph structure.

```
int MPI_Graph_map (
    MPI_Comm comm,          /* IN - Graph communicator */
    int n,                  /* IN - Vertices in graph */
    int *index,             /* IN - Index information */
    int *edge,              /* IN - Edge information */
    int *new_rank           /* OUT - New rank of process */
)
```

Collective function `MPI_Graph_map` attempts to optimize the placement of processes on processors, given the connections specified in a graph communicator. The new rank of the calling process is returned through the last parameter. This value is `MPI_UNDEFINED` if the calling process is not part of the graph communicator.

```
int MPI_Graph_neighbors (
    MPI_Comm comm,          /* IN - Graph communicator */
    int rank,               /* IN - Process rank */
    int max_neighbors,      /* IN - Max number of neighbors */
    int *neighbors          /* OUT - Ranks of neighbors */
)
```

Function `MPI_Graph_neighbors` returns through the last parameter the rank numbers of the processes that are neighbors of the specified process. (This list of neighbors is part of the `edge` array used to create the graph communicator.)

```
int MPI_Graph_neighbors_count (
    MPI_Comm comm,          /* IN - Graph communicator */
    int rank,               /* IN - Process rank */
    int *neighbors          /* OUT - Number of neighbors */
)
```

Function `MPI_Graph_neighbors_count` returns the number of neighbors in the specified communicator for the process having the specified rank.

```
int MPI_Graphdims_get (
    MPI_Comm comm,          /* IN - Graph communicator */
    int *vertices,          /* OUT - Vertices in the graph */
    int *edges              /* OUT - Edges in the graph */
)
```

Passed a graph communicator, MPI_Graphdims_get returns the number of vertices (processes) in the communicator and the number of directed edges (links) between these processes.

```
int MPI_Group_compare (
   MPI_Group group1,        /* IN - First process group */
   MPI_Group group2,        /* IN - Second process group */
   int *result              /* OUT - Result of comparison */
)
```

Function MPI_Group_compare compares two groups of processes. It returns MPI_IDENT if the two groups have the same processes and the same process ranking; MPI_SIMILAR if the two groups have the same processes but their rankings are different; and MPI_UNEQUAL otherwise.

```
int MPI_Group_difference (
   MPI_Group group1,          /* IN - First group */
   MPI_Group group2,          /* IN - Second group */
   MPI_Group *group_diff      /* OUT - Difference */
)
```

Function MPI_Group_difference, when passed two process groups, produces a new process group whose members are all processes in the first group that are not in the second group. The ordering of the processes is the same as in the first group.

```
int MPI_Group_excl (
   MPI_Group group, /* IN - Existing process group */
   int excl_num,    /* IN - Number of processes to exclude */
   int *excl_ranks, /* IN - Ranks of excluded processes */
   MPI_Group *new;  /* OUT - New group */
)
```

Function MPI_Group_excl creates a new group by removing processes with particular rank numbers from an existing group. The processes in the new group have the same order as in the original group.

```
int MPI_Group_free (
   MPI_Group *group           /* IN - Process group */
)
```

Function MPI_Group_free marks a group object for deallocation and changes the group handle to MPI_GROUP_NULL. The group object will not be deallocated until all operations using the group have completed.

```
int MPI_Group_incl (
   MPI_Group old;   /* IN - Existing process group */
   int new_size,    /* IN - Number of procs in new group */
```

```
    int *old_ranks,  /* IN - Order of procs in new group */
    MPI_Group *new;  /* OUT - New process group */
)
```

Function MPI_Group_incl produces a new group from an existing group. The new group may be smaller than the existing group. The size of the new group is specified by parameter new_size. Only the processes whose ranks appear in old_ranks are in the new group. The order in which rank numbers appear in old_ranks determines the processes' rankings in the new group. The first process identified in old_ranks has rank 0 in the new group. The last process identified in old_ranks has the highest rank in the new group.

```
int MPI_Group_intersection (
    MPI_Group group1,      /* IN - Group 1 */
    MPI_Group group2,      /* IN - Group 2 */
    MPI_Group *new_group   /* OUT - Intersection group */
)
```

Function MPI_Group_intersection produces a new group that is the intersection of two existing groups. The ordering of processes is as in the first group.

```
int MPI_Group_range_excl (
    MPI_Group group, /* IN - Existing process group */
    int n,           /* IN - Ranges to evaluate */
    int range[][3],  /* IN - Ranges of processes to exclude */
    MPI_Group *new   /* OUT - New process group */
)
```

Function MPI_Group_range_excl produces a new group from an old group and a set of n ranges. Each range consists of a first rank, a last rank, and a stride. For example, the range {5, 11, 3} represents processes with ranks 5, 8, and 11. The new group consists of processes in the original group that are not included in any of the ranges. The ordering of these processes is identical to their ordering in the original group.

```
int MPI_Group_range_incl (
    MPI_Group old,   /* IN - Existing process group */
    int n,           /* IN - Ranges to evaluate */
    int range[][3],  /* IN - Ranges of processes to include */
    MPI_Group *new   /* OUT - New process group */
)
```

Function MPI_Group_range_incl produces a new group from an old group and a set of n ranges. Each range consists of a first rank, a last rank, and a stride. For example, the range {5, 11, 3} represents processes with ranks 5, 8, and 11. The new group consists of only those processes included in one of the ranges. The ordering of these processes is identical to their ordering in the original group.

```
int MPI_Group_rank (
   MPI_Group group,        /* IN - Process group */
   int *rank               /* OUT - Rank of process */
)
```

Passed a process group handle, `MPI_Group_rank` returns the rank of the calling process in that group.

```
int MPI_Group_size (
   MPI_Group group,        /* IN - Process group */
   int *size               /* OUT - Size of group */
)
```

Passed a process group handle, `MPI_Group_size` returns the number of processes in the group.

```
int MPI_Group_translate_ranks (
   MPI_Group group1, /* IN - First group */
   int n,            /* IN - Number of ranks to compare */
   int *rank1,       /* IN - Valid ranks in first group */
   MPI_Group group2, /* IN - Second group */
   int *rank2        /* OUT - Ranks in second group */
)
```

Use function `MPI_Group_translate_ranks` to determine, for particular processes in one group, their ranks in a second group. Parameter `n` represents the number of comparisons to make. Array `rank1` lists ranks of processes in `group1`. The function returns through array `rank2` the corresponding ranks of these processes in `group2`. For example, suppose the process with rank 3 in `group1` has rank 4 in `group2`. Then if `rank1[i]= 3`, then `rank2[i]=4`.

```
int MPI_Group_union (
   MPI_Group group1,     /* IN - First group */
   MPI_Group group2,     /* IN - Second group */
   MPI_Group *new_group  /* OUT - Union of two groups */
)
```

Function `MPI_Group_union` returns a new group that is the union of the groups supplied as the first two parameters. The ordering of the processes is all elements of the first group followed by all elements of the second group that are not in the first group.

```
int MPI_Ibsend (
   void *buffer,         /* IN - Message buffer */
   int cnt,              /* IN - Elements in message */
   MPI_Datatype dtype,   /* IN - Element type */
   int dest,             /* IN - Rank of destination process */
   int tag,              /* IN - Message identifier */
```

```
   MPI_Comm comm,      /* IN - Communicator */
   MPI_Request *handle /* OUT - Handle to request */
)
```

Function `MPI_Ibsend` initiates an immediate (nonblocking) buffered send. Use the request handle to query the status of the send or wait for its completion. Because the call is nonblocking, do not access the message buffer until the send completes.

```
int MPI_Init (
   int *argc,    /* IN - First parameter to function main */
   char ***argv  /* IN - Second parameter to function main */
)
```

Function `MPI_Init` allows the parallel environment to be set up. An MPI program must call `MPI_Init` before any other MPI function. (The only exception is function `MPI_Initialized`.) Subsequent calls to `MPI_Init` are erroneous. Pass pointers to the same parameters `argc` and `argv` appearing in function `main`.

```
int MPI_Init_thread (
   int *argc,    /* IN - First parameter to function main */
   char ***argv, /* IN - Second parameter to function main */
   int desired,  /* IN - Desired level of thread support */
   int *provided /* OUT - Provided level of thread support */
)
```

Function `MPI_Init_thread` initializes MPI in the same way as `MPI_Init`. In addition, it initializes the thread environment. If `MPI_Init_thread` is called, then a call to `MPI_Init` is omitted. The first two arguments are the same as to `MPI_Init`. The third argument is the desired level of thread support:

1. `MPI_THREAD_SINGLE`—Single thread execution.
2. `MPI_THREAD_FUNNELED`—While the process may be multithreaded, only the main thread will make MPI calls.
3. `MPI_THREAD_SERIALIZED`—Multiple threads may make MPI calls, but these calls are serialized.
4. `MPI_THREAD_MULTIPLE`—Multiple threads may make MPI calls concurrently.

A higher number represents a higher level of support. The function returns through the last parameter the value of the support the system can provide.

Implementations of MPI are not required to support threads.

```
int MPI_Initialized (
   int* flag  /* OUT - Indicates if MPI has been initialized */
)
```

Function MPI_Initialized, when passed a pointer to an integer, sets the value of the integer to true if MPI_Init has been called, and false otherwise. This is the only function that may be called before MPI_Init.

```
int MPI_Intercomm_create (
   MPI_Comm local_comm,   /* IN - Local communicator */
   int local_leader,      /* IN - Rank of local "leader" */
   MPI_Comm remote_comm,  /* IN - Remote communicator */
   int remote_leader,     /* IN - Rank of remote "leader" */
   int tag,               /* IN - Intercomm identifier */
   MPI_Comm *new_comm     /* OUT - Inter-communicator */
)
```

The collective function MPI_Intercomm_create creates a new inter-communicator from an existing, local communicator (of which the process is a member) and a remote communicator (of which the process is not a member). Parameter tag is used to disambiguate messages associated with setting up the inter-communicator in the event that multiple inter-communicators are being constructed concurrently. The resulting inter-communicator still has the notion of "local" and "remote" groups. At least one member from each group (the leaders) have the ability to communicate with each other.

```
int MPI_Intercomm_merge (
   MPI_Comm inter,    /* IN - Handle to inter-communicator */
   int high,          /* IN - "High" group indicator */
   MPI_Comm *intra    /* OUT - Handle to intracommunicator */
)
```

Collective function MPI_Intercomm_merge converts an inter-communicator into an intracommunicator. All the processes in one group of the inter-communicator should set high to true, while all processes in the other group should set high to false. When determining the ranks of the processes in the new intracommunicator, the system orders the "low" processes before the "high" processes.

```
int MPI_Iprobe (
   int src,              /* IN - Rank of sending process */
   int tag,              /* IN - Incoming message tag */
   MPI_Comm comm,        /* IN - Communicator */
   int *flag,            /* OUT - Success flag */
   MPI_Status *status    /* OUT - Pointer to status object */
)
```

MPI_Iprobe is a nonblocking function that checks for an incoming message without actually receiving the message. When the function returns, flag is true if a message from the specified source process with the specified tag is ready to be received. Otherwise, flag is false when the function returns. If flag is true, information about the message can be retrieved through the status pointer. This

function is useful when you want to allocate a receive buffer based on the size of an incoming message. To allow any message source, use MPI_ANY_SOURCE. To allow any message tag, use MPI_ANY_TAG. If you want your program to block until the message is ready to be received, use function MPI_Probe.

```
int MPI_Irecv (
   void *buffer,          /* OUT - Address of receive buffer */
   int cnt,               /* IN - Elements to receive */
   MPI_Datatype dtype,    /* IN - Type of message elements */
   int src,               /* IN - Source process of message */
   int tag,               /* IN - Message ID */
   MPI_Comm comm,         /* IN - Communicator */
   MPI_Request *handle    /* OUT - Request handle */
)
```

MPI_Irecv implements a nonblocking, or immediate, receive. It posts a request for the receive to the run-time system, then returns control immediately to the calling function. Do not access the receive buffer until completing the receive with a call to MPI_Wait.

```
int MPI_Irsend (
   void *buffer,          /* IN - Message buffer */
   int cnt,               /* IN - Elements in message */
   MPI_Datatype dtype,    /* IN - Element type */
   int dest,              /* IN - Rank of destination process */
   int tag,               /* IN - Message identifier */
   MPI_Comm comm,         /* IN - Communicator */
   MPI_Request *handle    /* OUT - Handle to request */
)
```

Function MPI_Irsend initiates an immediate (nonblocking) ready send. Use the request handle to query the status of the send or wait for its completion. Because the call is nonblocking, do not access the message buffer until the send completes.

```
int MPI_Isend (
   void *buffer,          /* IN - Message buffer */
   int cnt,               /* IN - Elements in message */
   MPI_Datatype dtype,    /* IN - Type of elements */
   int dest,              /* IN - Destination process */
   int tag,               /* IN - Message identifier */
   MPI_Comm comm,         /* IN - Communicator */
   MPI_Request *handle    /* OUT - Request handle */
)
```

Function MPI_Isend is a nonblocking, or immediate, communication function. It posts the communication request to the run-time system and returns immediately to the calling procedure, having initialized a pointer to an opaque object

containing information about the pending send. You must complete the function by passing the handle to another MPI function, such as `MPI_Wait`. Until then, you should not modify the contents of the send buffer.

```
int MPI_Issend (
   void *buffer,          /* IN - Message buffer */
   int cnt,               /* IN - Elements in message */
   MPI_Datatype dtype,    /* IN - Element type */
   int dest,              /* IN - Rank of destination process */
   int tag,               /* IN - Message identifier */
   MPI_Comm comm,         /* IN - Communicator */
   MPI_Request *handle    /* OUT - Handle to request */
)
```

Function `MPI_Issend` initiates a synchronous mode immediate (nonblocking) send. Control does not return to the calling function until the corresponding receives have started. Use the request handle to query the status of the send or wait for its completion. Because the call is nonblocking, do not access the message buffer until the send completes.

```
int MPI_Keyval_create (
   MPI_Copy_function
      *copy_fn,       /* IN - Ptr to copy attribute function */
   MPI_Delete_function
      *del_fn,        /* IN - Ptr to delete attribute func */
   int *key,          /* OUT - Ptr to attribute key */
   void *extra        /* IN - Extra info for callbacks */
)
```

Function `MPI_Keyval_create` creates a new attribute key, identified by `key`.

```
int MPI_Keyval_free (
   int *keyval                    /* IN - Key value */
)
```

Function `MPI_Keyval_free` marks an integer key value for deallocation and sets the value of `keyval` to `MPI_KEYVAL_INVALID`.

```
int MPI_Op_create (
   MPI_User_function
      *assoc_func,     /* IN - Associative function */
   int commutative,    /* IN - Commutativity flag (logical) */
   MPI_Op *op          /* OUT - Op handle */
)
```

You can define your own global reduction operation and use function `MPI_Op_create` to bind it to an op handle that can be used in calls to `MPI_Reduce`, `MPI_Allreduce`, `MPI_Reduce_scatter`, and `MPI_Scan`. The

global operation must be associative. If the operation is commutative as well, make the second parameter true. The function returns the op handle through the third parameter.

The ANSI-C prototype for the function performing your global operation is

```
typedef void MPI_User_function(void *in_vector,
   void *in_out_vector, int *length, MPI_Datatype *dtype);
```

Let $u[0], u[1], \ldots, u[length - 1]$ represent the elements of `in_vector` when the function is invoked; $v[0], v[1], \ldots, v[length - 1]$ represent the elements of `in_out_vector` when the function is invoked; and $w[0], w[1], \ldots, w[length - 1]$ represent the elements of `in_out_vector` when the function returns. Finally, let $\oplus$ represent the associative operation computed by the function. Then the elements of w should be computed as follows: $w[i] = u[i] \oplus v[i]$, for all $i < length$. In other words, your function overwrites the values in `in_out_vector` with the result.

```
int MPI_Op_free (
   MPI_Op *op    /* IN - Handle to a user-defined operation */
)
```

Function `MPI_Op_free` marks a user-defined operation for deallocation. It changes the value of `op` to `MPI_OP_NULL`.

```
int MPI_Pack (
   void *in_buffer,      /* IN - Original message buffer */
   int elements,         /* IN - Elements in message buffer */
   MPI_Datatype dtype,   /* IN - Type of elements */
   void *out_buffer,     /* OUT - Packed message buffer */
   int out_size,         /* IN - Bytes in out_buffer */
   int *offset,          /* IN/OUT - Index in out_buffer where
                                packing starts/ends */
   MPI_Comm comm         /* IN - Communicator used in
                                subsequent send */
)
```

You can use function `MPI_Pack` to pack noncontiguous data into a contiguous buffer before sending it. After the message is received, it must be unpacked. An alternative to packing and unpacking is to use derived datatypes. Another use of packing and unpacking is to avoid system buffering.

```
int MPI_Pack_size (
   int cnt,              /* IN - Elements in message */
   MPI_Datatype dtype,   /* IN - Element type */
   MPI_Comm comm,        /* IN - Communicator */
   int *bound            /* OUT - Upper bound on packed
                                message size */
)
```

Function MPI_Pack_size computes an upper bound on the number of bytes a packed message will occupy. By calling this function before MPI_Pack, you can determine how large to make the buffer containing the packed message.

```
int MPI_Probe (
   int src,              /* IN - Rank of message source */
   int tag,              /* IN - Incoming message tag */
   MPI_Comm comm,        /* IN - Communicator */
   MPI_Status *status    /* OUT - Pointer to status object */
)
```

Function MPI_Probe checks for an incoming message without actually receiving the message. It is useful when you want to allocate a receive buffer based on the size of an incoming message. To allow any message source, use MPI_ANY_SOURCE. To allow any message tag, use MPI_ANY_TAG. This function blocks until a message matching the source and tag has arrived. To check for the existence of a message without blocking, use function MPI_Iprobe.

```
int MPI_Recv (
   void* buffer,         /* OUT - Receive buffer */
   int cnt,              /* IN - Max number of elements
                                 to receive */
   MPI_Datatype dtype,   /* IN - Type of message elements */
   int src,              /* IN - Source process of message */
   int tag,              /* IN - Message ID */
   MPI_Comm comm,        /* IN - Communicator */
   MPI_Status *status    /* OUT - Result of receive */
)
```

MPI_Recv implements a blocking receive. Assuming the receive is successful, when control returns from MPI_Recv, buffer points to the received message.

```
int MPI_Recv_init (
   void *buffer,         /* OUT - Receive buffer */
   int cnt,              /* IN - Max number of elements
                                 to receive */
   MPI_Datatype dtype,   /* IN - Type of element */
   int src,              /* IN - Rank of message source */
   int tag,              /* IN - Message identification */
   MPI_Comm comm,        /* IN - Communicator */
   MPI_Request *handle   /* OUT - Request handle */
)
```

Function MPI_Recv_init creates a persistent communication request for a standard mode receive operation. It is useful when your program repeatedly calls the receive function with identical arguments. The receive is actually initiated with a call to MPI_Start.

```
int MPI_Reduce (
   void *send_buffer,    /* IN - Send buffer */
   void *recv_buffer,    /* OUT - Receive buffer. Only root
                            process gets results. */
   int cnt,              /* IN - Number elements to reduce */
   MPI_Datatype dtype,   /* IN - Element type */
   MPI_Op op,            /* IN - Reduction operator */
   int root,             /* IN - Rank of root process */
   MPI_Comm comm         /* IN - Communicator */
)
```

`MPI_Reduce` is a collective communication function that performs `cnt` reductions. When the function returns, the process with rank `root` has the results of the reductions. Use `MPI_Allreduce` if you want all processes to have the results of the reductions.

```
int MPI_Reduce_scatter (
   void *send_buffer,  /* IN - Send buffer */
   void *recv_buffer,  /* OUT - Receive buffer */
   int *recv_cnts,     /* IN - Group-sized array. Entry j
                             is the number of result
                             elements to send process j. */
   MPI_Datatype dtype, /* IN - Element type */
   MPI_Op op,          /* IN - MPI reduction operator */
   MPI_Comm comm       /* IN - Communicator */
)
```

Function `MPI_Reduce_scatter` is a collective communication function that performs a reduction and then scatters the resulting elements. The number of elements reduced is the sum of the values in array `recv_cnts`.

```
int MPI_Request_free (
   MPI_Request *handle   /* IN - Message request handle */
)
```

Function `MPI_Request_free` requests deallocation of an `MPI_Request` object associated with a persistent communication. Any outstanding communication associated with the request object will finish before the system deallocates the object.

```
int MPI_Rsend (
   void *buffer,         /* IN - Message buffer */
   int cnt,              /* IN - Elements in message */
   MPI_Datatype dtype,   /* IN - Type of elements */
   int dest,             /* IN - Destination process */
   int tag,              /* IN - Message identifier */
   MPI_Comm comm         /* IN - Communicator */
);
```

Function MPI_Rsend implements a ready mode send. In the ready mode, a send may only be started if the matching receive is already posted. If the matching receive has not already been posted, it is an error condition. The use of ready mode sends can improve performance on some systems by eliminating a handshake operation. When this function returns, the send buffer may be reused.

```
int MPI_Rsend_init (
   void *buffer,          /* IN - Send buffer */
   int cnt,               /* IN - Elements in message */
   MPI_Datatype dtype,    /* IN - Type of elements */
   int dest,              /* IN - Rank of destination */
   int tag,               /* IN - Message identifier */
   MPI_Comm comm,         /* IN - Communicator */
   MPI_Request *handle    /* OUT - Request handle */
)
```

MPI_Rsend_init creates a persistent communication request for a ready mode send. It is useful when your program repeatedly executes a communication with the same argument list. A call to MPI_Start actually initiates the message send.

```
int MPI_Scan (
   void *send_buffer,     /* IN - Send buffer */
   void *recv_buffer,     /* OUT - Receive buffer */
   int cnt,               /* IN - Size of send buffer */
   MPI_Datatype dtype,    /* IN - Type of sent elements */
   MPI_Op op,             /* IN - MPI operator */
   MPI_Comm comm          /* IN - Communicator */
)
```

MPI_Scan performs a parallel prefix operator. After the function has returned, each element in recv_buffer is the result of using the MPI operator op to combine the similar recv_buffer elements in all processes having equal or lower rank.

```
int MPI_Scatter (
   void *send_buffer, /* IN - Send buffer */
   int send_cnt,      /* IN - Elements sent each process */
   MPI_Datatype
      send_dtype,     /* IN - Type of sent elements */
   void *recv_buffer, /* OUT - Address of receive buffer */
   int recv_cnt,      /* IN - Number of elements this process
                            receives */
   MPI_Datatype
      recv_dtype,     /* IN - Type of received elements */
   int root,          /* IN - Rank of sending process */
   MPI_Comm comm      /* IN - Communicator */
)
```

`MPI_Scatter` is a collective communication function that performs a scatter: a group of elements held by the root process is divided into equal-sized chunks, and one chunk is sent to every process in the communicator (including the root). Use the more general function `MPI_Scatterv` if the group of elements is divided into chunks of varying size.

```
int MPI_Scatterv (
   void *send_buffer, /* IN - Send buffer */
   int *send_cnts,    /* IN - Number of elements to send to
                            each process */
   int *send_disp,    /* IN - Element i is the offset in
                            send_buffer of the first data
                            element going to process i */
   MPI_Datatype
      send_dtype,     /* IN - Type of sent elements */
   void *recv_buffer, /* OUT - Receive buffer */
   int recv_cnt,      /* IN - Number of elements this
                            process will receive */
   MPI_Datatype
      recv_dtype,     /* IN - Type of received elements */
   int root,          /* IN - Rank of sending process */
   MPI_Comm comm      /* IN - Communicator */
)
```

`MPI_Scatterv` is a collective communication function that performs a scatter: a group of elements held by the root process is divided into chunks, and one chunk is sent to every process in the communicator (including the root). The array `send_cnts` indicates the number of elements going to each process; array `send_disp` contains the offset inside `send_buffer` of the chunk destined for each process. Use the simpler function `MPI_Scatter` if the group of elements is divided into equal-sized chunks and the chunks are distributed in process rank order.

```
int MPI_Send (
   void *buffer,        /* IN - Message buffer */
   int cnt,             /* IN - Elements in message */
   MPI_Datatype dtype,  /* IN - Type of elements */
   int dest,            /* IN - Destination process */
   int tag,             /* IN - Message identifier */
   MPI_Comm comm        /* IN - Communicator */
)
```

Function `MPI_Send` implements a blocking send operation: when the function returns, the message buffer may be immediately reused. The MPI run-time system decides whether to copy the message into a system buffer or copy the message directly into the matching receive buffer. If the run-time system does not choose to buffer the outgoing message, the call to `MPI_Send` will not return until the message has been sent to the receiving process.

```
int MPI_Send_init (
   void *buffer,        /* IN - Send buffer */
   int cnt,             /* IN - Elements in message */
   MPI_Datatype dtype,  /* IN - Type of elements */
   int dest,            /* IN - Rank of destination */
   int tag,             /* IN - Message identifier */
   MPI_Comm comm,       /* IN - Communicator */
   MPI_Request *handle /* OUT - Request handle */
)
```

`MPI_Send_init` creates a persistent communication request for a standard mode send. It is useful when your program repeatedly executes a communication with the same argument list. A call to `MPI_Start` actually initiates the message send.

```
int MPI_Sendrecv (
   void *send_buffer,/* IN - Send buffer */
   int send_cnt,     /* IN - Elements to send */
   MPI_Datatype
      send_dtype,    /* IN - Outgoing message element type */
   int dest,         /* IN - Destination process for
                            outgoing message */
   int send_tag,     /* IN - Outgoing message ID */
   void *recv_buffer,/* OUT - Receive buffer */
   int recv_cnt,     /* IN - Elements to receive */
   MPI_Datatype
      recv_dtype,    /* IN - Incoming message element type */
   int src,          /* IN - Source process of incoming
                            message */
   int recv_tag,     /* IN - Incoming message ID */
   MPI_Comm comm,    /* IN - Communicator */
   MPI_Status *status/* OUT - Status of received message */
)
```

`MPI_Sendrecv` combines in a single function call the sending of a message to one process and the receiving of a message from that process or another process. It is particularly useful when processes form a ring and each process is sending a message to its neighbor. If ordinary blocking sends and receives are used, careful choreography is needed to prevent deadlock. Replacing two individual send-receive function calls with a single call to `MPI_Sendrecv` guarantees a deadlock will not occur. Function `MPI_Sendrecv` is a blocking send and receive operation. The send and receive buffers must be disjoint. Use `MPI_Sendrecv_replace` if you want the send and receive buffers to be the same.

```
int MPI_Sendrecv_replace (
   void *buffer,          /* IN/OUT - Message buffer */
   int cnt,               /* IN - Elements in message */
   MPI_Datatype dtype,    /* IN - Type of message elements */
   int dest,              /* IN - Destination process */
   int send_tag,          /* IN - Sent message ID */
   int src,               /* IN - Source process */
   int recv_tag,          /* IN - Received message ID */
   MPI_Comm comm,         /* IN - Communicator */
   MPI_Status *status     /* OUT - Received message status */
)
```

Function MPI_Sendrecv_replace performs a blocking send and receive. It is similar to MPI_Sendrecv, except that the buffer containing the message to be sent is also where the received message will be stored. The length of both the sent and received messages must be identical.

```
int MPI_Ssend (
   void *buffer,          /* IN - Send buffer */
   int cnt,               /* IN - Number of elements to send */
   MPI_Datatype dtype,    /* IN - Type of elements */
   int dest,              /* IN - Destination process */
   int tag,               /* IN - Message tag */
   MPI_Comm comm          /* IN - Communicator */
)
```

Function MPI_Ssend implements a synchronous mode send operation, in which the send function successfully finishes only when a matching receive has been posted and the receiving process has begun to receive the message. Completion of MPI_Ssend indicates the send buffer can be reused.

```
int MPI_Ssend_init (
   void *buffer,          /* IN - Send buffer */
   int cnt,               /* IN - Elements in message */
   MPI_Datatype dtype,    /* IN - Type of elements */
   int dest,              /* IN - Rank of destination */
   int tag,               /* IN - Message identifier */
   MPI_Comm comm,         /* IN - Communicator */
   MPI_Request *handle    /* IN - Request handle */
)
```

MPI_Ssend_init creates a persistent communication request for a synchronous mode send. It is useful when your program repeatedly executes a communication with the same argument list. A call to MPI_Start actually initiates the message send.

```
int MPI_Start (
   MPI_Request *handle    /* IN - Request handle */
)
```

Call function MPI_Start to initiate the persistent communication request (either a send or a receive) represented by the handle.

```
int MPI_Startall (
   int size,              /* IN - Elements in request_array */
   MPI_Request *requests /* IN - Array of pointers to
                              communication objects */
)
```

Calling function MPI_Startall with an array of size communication handles is equivalent to a series of size calls to MPI_Start that pass the handles in requests in some arbitrary order.

```
int MPI_Test (
   MPI_Request *handle, /* IN - Persistent request handle */
   int *flag,           /* OUT - Completion flag */
   MPI_Status *status   /* OUT - Results of communication */
)
```

Use function MPI_Test to determine if the operation associated with a communication request has been completed. The function returns the value true for flag if the operation has been completed, and false otherwise. If a receive operation has ended, you can access status to find the message source (status->MPI_SOURCE), message tag (status->MPI_TAG), and error code (status->MPI_ERROR).

```
int MPI_Test_cancelled (
   MPI_Status *handle,     /* IN - Communication handle */
   int *flag               /* OUT - Result flag */
)
```

Function MPI_Test_cancelled, when passed the handle to a communication object, returns through flag the value true if the communication was successfully cancelled, and false otherwise.

```
int MPI_Testall (
   int cnt,               /* IN - Requests to test */
   MPI_Request *handle,   /* IN - Array of request handles */
   int *flag,             /* OUT - Result flag */
   MPI_Status *status     /* OUT - Array of status info */
)
```

When passed handles to cnt communication request objects, function MPI_Testall returns a value of true through flag if and only if all of the communications have ended. If flag is true, then the elements of array status are set to reflect the outcomes of the communications.

```
int MPI_Testany (
   int cnt,             /* IN - Number of requests to check */
   MPI_Request *handle,/* IN - Handles to request objects */

   int *index,          /* OUT - Index of a completed
                               communication */
   int *flag,           /* OUT - Result flag */
   MPI_Status *status   /* OUT - Status information */
)
```

You can use function MPI_Testany to check to see if any of a list of cnt communications have ended. If flag is false when the function returns, none of the communications have been completed. If flag is true, then at least one of the communications has been completed. The value of index indicates the position in array handle of the handle to the completed communication. If the completed communication is a receive, additional information about it is available through status.

```
int MPI_Testsome (
   int in_cnt,          /* IN - Number of requests to test */
   MPI_Request
      *handlearray,     /* IN - Array of request handles */
   int *out_cnt,        /* OUT - Number of completed requests */
   int *index_array,    /* OUT - Array of request indices */
   MPI_Status
      *status_array     /* OUT - Array of status records */
)
```

Function MPI_Testsome returns information on all completed communications. Array index_array indicates which request handles in handle-array correspond to completed communications. Additional information about receives is available in status_array.

```
int MPI_Topo_test (
   MPI_Comm comm,   /* IN - Communicator */
   int *topology    /* OUT - Communicator's topology */
)
```

Function MPI_Topo_test returns the topology type of a communicator. The possible return values of topology are MPI_GRAPH for a graph topology, MPI_CART for a Cartesian topology, and MPI_UNDEFINED for no topology.

```
int MPI_Type_commit (
   MPI_Datatype *dtype  /* IN - Derived datatype object */
)
```

Function MPI_Type_commit commits a derived datatype so that it can be used in communication operations. See MPI_Type_free.

```
int MPI_Type_contiguous (
   int cnt,                  /* IN - Copies to make */
   MPI_Datatype old_dtype,   /* IN - Old datatype */
   MPI_Datatype *new_dtype   /* OUT - New datatype */
)
```

Function MPI_Type_contiguous creates a new datatype consisting of cnt copies of old_dtype concatenated together.

```
int MPI_Type_count (
   MPI_Datatype dtype, /* IN - Datatype */
   int *cnt            /* OUT - Top-level entry count */
)
```

Function MPI_Type_count returns the number of "top-level" entries in datatype dtype.

```
int MPI_Type_extent (
   MPI_Datatype dtype,     /* IN - Datatype */
   MPI_Aint *extent        /* OUT - Extent of dtype */
)
```

Function MPI_Type_extent returns the extent of a datatype dtype. A datatype's extent is the number of bytes a single instance of the datatype would occupy in a message. It is equal to the number of bytes occupied by the datatype's elements, rounded up to satisfy the underlying hardware's data alignment requirements.

```
int MPI_Type_free (
   MPI_Datatype *dtype     /* IN - Derived datatype */
)
```

Function MPI_Type_free marks for deallocation the datatype object associated with dtype and sets dtype to MPI_DATATYPE_NULL. Any outstanding communications involving dtype will end normally. See MPI_Type_commit.

```
int MPI_Type_hindexed (
   int cnt,                /* IN - Number of blocks */
   int *block_len_array,   /* IN - Elements in each block */
   MPI_Aint *disp_array,   /* IN - Block byte displacements */
   MPI_Datatype old,       /* IN - Handle to old datatype */
   MPI_Datatype *new       /* OUT - Handle to new datatype */
)
```

Function MPI_Type_hindexed is identical to function MPI_Type_indexed, except that the block displacements given in disp_array are given in bytes.

```
int MPI_Type_hvector (
   int cnt,              /* IN - Number of blocks */
   int len,              /* IN - Number of elements per block */
   MPI_Aint stride,      /* IN - Displacement in bytes between
                                 start of each block */
   MPI_Datatype old,     /* IN - Handle to old datatype */
   MPI_Datatype *new     /* OUT - Handle to new datatype */
)
```

Function `MPI_Type_hvector` constructs a new derived datatype by replicating an existing datatype. The new datatype consists of `cnt` blocks. Each block contains `block_length` copies of `old`. The distance between blocks (measured in bytes) is `stride`.

```
int MPI_Type_indexed (
   int cnt,              /* IN - Number of blocks in 'new' */
   int *block_len,       /* IN - Array indicating copies of old
                                 datatype in each block */
   int *disp,            /* IN - Block displacements array */
   MPI_Datatype old,     /* IN - Handle to old datatype */
   MPI_Datatype *new     /* OUT - Handle to new datatype */
)
```

Function `MPI_Type_indexed` takes a sequence of one or more blocks, each consisting of one or more copies of an old datatype `old`, and concatenates them together to form a derived datatype `new`. Integer `cnt` is the number of blocks to concatenate. Array `block_len` gives the number of copies of `old` in each block. Array `disp` gives the displacement of each block in multiples of the extent of `old`.

```
int MPI_Type_lb (
   MPI_Datatype dtype, /* IN - Handle to datatype */
   MPI_Aint *lb        /* OUT - Lower bound's displacement */
)
```

Function `MPI_Type_lb` returns the displacement in bytes of the lower bound of datatype `dtype` from the origin.

```
int MPI_Type_size (
   MPI_Datatype dtype, /* IN - Handle to datatype */
   int *size           /* OUT - Total size of type
                              signature entries */
)
```

The **type signature** of a datatype is the sequence of basic types it contains. Function `MPI_Type_size` returns the number of bytes occupied by the entries in the type signature of `dtype`. It is equal to the number of bytes of data in a message containing one element of the datatype.

```
int MPI_Type_struct (
   int cnt,              /* IN - Number of blocks in 'new' */
   int *block_len,       /* IN - Elements in each block */
   MPI_Aint *disp,       /* IN - Displacement of each block */
   MPI_Datatype *dtype,  /* IN - Array of datatype handles */
   MPI_Datatype *new     /* OUT - Handle to new datatype */
)
```

Function `MPI_Type_struct` constructs a new datatype consisting of `cnt` blocks. Each block `i` contains `block_len[i]` copies of the datatype corresponding to handle `dtype[i]`. The displacement of each block `i` is indicated by `disp[i]`.

```
int MPI_Type_ub (
   MPI_Datatype dtype, /* IN - Handle to datatype */
   MPI_Aint *ub        /* OUT - Upper bound's displacement */
)
```

Function `MPI_Type_ub` returns the displacement in bytes of the upper bound of datatype `dtype` from the origin.

```
int MPI_Type_vector (
   int cnt,                  /* IN - Number of blocks */
   int block_length,         /* IN - Elements in each block */
   int stride,               /* IN - Elements between start
                                    of each block */
   MPI_Datatype old_dtype,   /* IN - Handle to old datatype */
   MPI_Datatype *new_dtype,  /* OUT - New datatype's handle */
)
```

Function `MPI_Type_vector` constructs a new derived datatype by replicating an existing datatype. The new datatype consists of `cnt` blocks. Each block contains `block_length` copies of `old_dtype`. The distance between blocks (measured in terms of multiples of the extent of `old_dtype`) is `stride`.

```
int MPI_Unpack (
   void *in_buffer,     /* IN - Input buffer */
   int len,             /* IN - Length of input buffer */
   int *position,       /* IN/OUT - Position in 'in_buffer' */
   void *out_buffer,    /* OUT - Output buffer */
   int out_cnt,         /* IN - Number of items to unpack */
   MPI_Datatype dtype,  /* IN - Handle to
   MPI_Comm comm        /* IN - Communicator */
)
```

Function `MPI_Unpack` unpacks a message `len` bytes long from `in_buffer` into `out_buffer`.

At the beginning of the function's execution, `position` is the index in `in_buffer` of the beginning of the packed message. When the function returns,

position is the index of the first byte after the message that was unpacked. The position parameter allows several messages to be packed into a single **packing unit**, sent, and then unpacked individually.

```
int MPI_Wait (
   MPI_Request *handle,    /* IN - Request handle */
   MPI_Status *status      /* OUT - Result of communication */
)
```

Function MPI_Wait completes any nonblocking operation. If the operation was a send, this function waits until the message has been either buffered or sent by the run-time system. At this point the send buffer may be reused. If the operation was a receive, it waits until the message has been copied into the receive buffer.

```
int MPI_Waitall (
   int cnt,              /* IN - Number of comms to wait on */
   MPI_Request *handle,  /* IN - Request handles array */
   MPI_Status *status    /* OUT - Status of completed comms */
)
```

Function MPI_Waitall blocks until all cnt of the communication operations associated with the handles stored in handle_array have ended. When the function returns, the status array contains information about all of the completed communications.

```
int MPI_Waitany (
   int cnt,              /* IN - Number of comms to check */
   MPI_Request *handle,  /* IN - Array of request handles */
   int *index,           /* OUT - Index of completed comm */
   MPI_Status *status    /* OUT - Status of completed comm */
)
```

Function MPI_Waitany blocks until one of the specified communications operations has been completed. When it returns, the value of index is the index into handle of the completed communication, and status points to the status record of the completed communication.

```
int MPI_Waitsome (
   int in_cnt,           /* IN - Number of comms to check */
   MPI_Request *handle,  /* IN - Array of request handles */
   int *out_cnt,         /* OUT - Number of completed ops */
   int *index_array,     /* OUT - Array of completed ops */
   MPI_Status *status    /* OUT - Array of status info */
)
```

Function MPI_Waitsome blocks until one or more of the specified communication operations have ended. When it returns, the value of out_cnt is the number of operations that have been completed. The first out_cnt elements of index are the indices in handle of the completed operations, and the first out elements of status are the completion status objects for these communications.

```
double MPI_Wtick (void)
```

Function MPI_Wtick returns a double-precision floating-point number indicating the number of seconds between ticks of the clock used by function MPI_Wtime. For example, if the clock is incremented every microsecond, function MPI_Wtick should return the value 10^{-6}.

```
double MPI_Wtime (void)
```

Function MPI_Wtime returns a double-precision floating-point number representing the number of seconds since some time in the past. The definition of "some time in the past" is guaranteed not to change during the life of a process. Hence the elapsed time of a block of code can be determined by calling MPI_Wtime before it and after it and computing the difference.

APPENDIX B

Utility Functions

B.1 HEADER FILE MyMPI.h

```
/*   MyMPI.h
 *
 *   Header file for a library of matrix/vector
 *   input/output/redistribution functions.
 *
 *   Programmed by Michael J. Quinn
 *
 *   Last modification: 4 September 2002
 */

/************************** MACROS ***************************/

#define DATA_MSG            0
#define PROMPT_MSG          1
#define RESPONSE_MSG        2

#define OPEN_FILE_ERROR    -1
#define MALLOC_ERROR       -2
#define TYPE_ERROR         -3

#define MIN(a,b)            ((a)<(b)?(a):(b))
#define BLOCK_LOW(id,p,n)   ((id)*(n)/(p))
#define BLOCK_HIGH(id,p,n)  (BLOCK_LOW((id)+1,p,n)-1)
#define BLOCK_SIZE(id,p,n)  \
                    (BLOCK_HIGH(id,p,n)-BLOCK_LOW(id,p,n)+1)
#define BLOCK_OWNER(j,p,n)  (((p)*((j)+1)-1)/(n))
#define PTR_SIZE            (sizeof(void*))
#define CEILING(i,j)        (((i)+(j)-1)/(j))
```

```
/****************** MISCELLANEOUS FUNCTIONS *****************/

void  terminate (int, char *);

/*************** DATA DISTRIBUTION FUNCTIONS ***************/

void replicate_block_vector (void *, int, void *,
        MPI_Datatype, MPI_Comm);
void create_mixed_xfer_arrays (int, int, int, int**, int**);
void create_uniform_xfer_arrays (int, int, int, int**,int**);

/****************** INPUT FUNCTIONS ************************/

void read_checkerboard_matrix (char *, void ***, void **,
        MPI_Datatype, int *, int *, MPI_Comm);
void read_col_striped_matrix (char *, void ***, void **,
        MPI_Datatype, int *, int *, MPI_Comm);
void read_row_striped_matrix (char *, void ***, void **,
        MPI_Datatype, int *, int *, MPI_Comm);
void read_block_vector (char *, void **, MPI_Datatype,
        int *, MPI_Comm);
void read_replicated_vector (char *, void **, MPI_Datatype,
        int *, MPI_Comm);

/****************** OUTPUT FUNCTIONS ************************/

void print_checkerboard_matrix (void **, MPI_Datatype, int,
        int, MPI_Comm);
void print_col_striped_matrix (void **, MPI_Datatype, int,
        int, MPI_Comm);
void print_row_striped_matrix (void **, MPI_Datatype, int,
        int, MPI_Comm);
void print_block_vector (void *, MPI_Datatype, int,
        MPI_Comm);
void print_replicated_vector (void *, MPI_Datatype, int,
        MPI_Comm);
```

B.2 SOURCE FILE MyMPI.c

```
/*
 *   MyMPI.c -- A library of matrix/vector
 *   input/output/redistribution functions
 *
 *   Programmed by Michael J. Quinn
 *
 *   Last modification: 4 September 2002
 */
```

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
#include "MyMPI.h"


/****************** MISCELLANEOUS FUNCTIONS *****************/

/*
 *   Given MPI_Datatype 't', function 'get_size' returns the
 *   size of a single datum of that data type.
 */


int get_size (MPI_Datatype t) {
   if (t == MPI_BYTE) return sizeof(char);
   if (t == MPI_DOUBLE) return sizeof(double);
   if (t == MPI_FLOAT) return sizeof(float);
   if (t == MPI_INT) return sizeof(int);
   printf ("Error: Unrecognized argument to 'get_size'\n");
   fflush (stdout);
   MPI_Abort (MPI_COMM_WORLD, TYPE_ERROR);
}



/*
 *   Function 'my_malloc' is called when a process wants
 *   to allocate some space from the heap. If the memory
 *   allocation fails, the process prints an error message
 *   and then aborts execution of the program.
 */


void *my_malloc (
   int id,     /* IN - Process rank */
   int bytes)  /* IN - Bytes to allocate */
{
   void *buffer;
   if ((buffer = malloc ((size_t) bytes)) == NULL) {
      printf ("Error: Malloc failed for process %d\n", id);
      fflush (stdout);
      MPI_Abort (MPI_COMM_WORLD, MALLOC_ERROR);
   }
   return buffer;
}
```

```
/*
 *   Function 'terminate' is called when the program should
 *   not continue execution, due to an error condition that
 *   all of the processes are aware of. Process 0 prints the
 *   error message passed as an argument to the function.
 *
 *   All processes must invoke this function together!
 */

void terminate (
   int   id,              /* IN - Process rank */
   char *error_message) /* IN - Message to print */
{
   if (!id) {
      printf ("Error: %s\n", error_message);
      fflush (stdout);
   }
   MPI_Finalize();
   exit (-1);
}


/************ DATA DISTRIBUTION FUNCTIONS ******************/

/*
 *   This function creates the count and displacement arrays
 *   needed by scatter and gather functions, when the number
 *   of elements send/received to/from other processes
 *   varies.
 */

void create_mixed_xfer_arrays (
   int id,        /* IN - Process rank */
   int p,         /* IN - Number of processes */
   int n,         /* IN - Total number of elements */
   int **count,   /* OUT - Array of counts */
   int **disp)    /* OUT - Array of displacements */
{

   int i;

   *count = my_malloc (id, p * sizeof(int));
   *disp = my_malloc (id, p * sizeof(int));
   (*count)[0] = BLOCK_SIZE(0,p,n);
   (*disp)[0] = 0;
   for (i = 1; i < p; i++) {
      (*disp)[i] = (*disp)[i-1] + (*count)[i-1];
      (*count)[i] = BLOCK_SIZE(i,p,n);
   }
}
```

```
/*
 *   This function creates the count and displacement arrays
 *   needed in an all-to-all exchange, when a process gets
 *   the same number of elements from every other process.
 */

void create_uniform_xfer_arrays (
   int id,          /* IN - Process rank */
   int p,           /* IN - Number of processes */
   int n,           /* IN - Number of elements */
   int **count,     /* OUT - Array of counts */
   int **disp)      /* OUT - Array of displacements */
{

   int i;

   *count = my_malloc (id, p * sizeof(int));
   *disp = my_malloc (id, p * sizeof(int));
   (*count)[0] = BLOCK_SIZE(id,p,n);
   (*disp)[0] = 0;
   for (i = 1; i < p; i++) {
      (*disp)[i] = (*disp)[i-1] + (*count)[i-1];
      (*count)[i] = BLOCK_SIZE(id,p,n);
   }
}

/*
 *   This function is used to transform a vector from a
 *   block distribution to a replicated distribution within a
 *   communicator.
 */

void replicate_block_vector (
   void        *ablock,  /* IN - Block-distributed vector */
   int          n,       /* IN - Elements in vector */
   void        *arep,    /* OUT - Replicated vector */
   MPI_Datatype dtype,   /* IN - Element type */
   MPI_Comm     comm)    /* IN - Communicator */
{
   int *cnt;  /* Elements contributed by each process */
   int *disp; /* Displacement in concatenated array */
   int id;    /* Process id */
   int p;     /* Processes in communicator */

   MPI_Comm_size (comm, &p);
   MPI_Comm_rank (comm, &id);
   create_mixed_xfer_arrays (id, p, n, &cnt, &disp);
   MPI_Allgatherv (ablock, cnt[id], dtype, arep, cnt,
                   disp, dtype, comm);
```

```
   free (cnt);
   free (disp);
}

/********************* INPUT FUNCTIONS *********************/

/*
 *   Function 'read_checkerboard_matrix' reads a matrix from
 *   a file. The first two elements of the file are integers
 *   whose values are the dimensions of the matrix ('m' rows
 *   and 'n' columns). What follows are 'm'*'n' values
 *   representing the matrix elements stored in row-major
 *   order.  This function allocates blocks of the matrix to
 *   the MPI processes.
 *
 *   The number of processes must be a square number.
 */

void read_checkerboard_matrix (
   char *s,              /* IN - File name */
   void ***subs,         /* OUT - 2D array */
   void **storage,       /* OUT - Array elements */
   MPI_Datatype dtype,   /* IN - Element type */
   int *m,               /* OUT - Array rows */
   int *n,               /* OUT - Array cols */
   MPI_Comm grid_comm)   /* IN - Communicator */
{
   void      *buffer;          /* File buffer */
   int        coords[2];       /* Coords of proc receiving
                                  next row of matrix */
   int        datum_size;      /* Bytes per element */
   int        dest_id;         /* Rank of receiving proc */
   int        grid_coord[2];   /* Process coords */
   int        grid_id;         /* Process rank */
   int        grid_period[2]; /* Wraparound */
   int        grid_size[2];    /* Dimensions of grid */
   int        i, j, k;
   FILE      *infileptr;       /* Input file pointer */
   void      *laddr;           /* Used when proc 0 gets row */
   int        local_cols;      /* Matrix cols on this proc */
   int        local_rows;      /* Matrix rows on this proc */
   void     **lptr;            /* Pointer into 'subs' */
   int        p;               /* Number of processes */
   void      *raddr;           /* Address of first element
                                  to send */
   void      *rptr;            /* Pointer into 'storage' */
   MPI_Status status;          /* Results of read */
```

```
MPI_Comm_rank (grid_comm, &grid_id);
MPI_Comm_size (grid_comm, &p);
datum_size = get_size (dtype);

/* Process 0 opens file, gets number of rows and
   number of cols, and broadcasts this information
   to the other processes. */

if (grid_id == 0) {
   infileptr = fopen (s, "r");
   if (infileptr == NULL) *m = 0;
   else {
      fread (m, sizeof(int), 1, infileptr);
      fread (n, sizeof(int), 1, infileptr);
   }
}
MPI_Bcast (m, 1, MPI_INT, 0, grid_comm);

if (!(*m)) MPI_Abort (MPI_COMM_WORLD, OPEN_FILE_ERROR);

MPI_Bcast (n, 1, MPI_INT, 0, grid_comm);

/* Each process determines the size of the submatrix
   it is responsible for. */

MPI_Cart_get (grid_comm, 2, grid_size, grid_period,
   grid_coord);

local_rows = BLOCK_SIZE(grid_coord[0],grid_size[0],*m);
local_cols = BLOCK_SIZE(grid_coord[1],grid_size[1],*n);

/* Dynamically allocate two-dimensional matrix 'subs' */

*storage = my_malloc (grid_id,
   local_rows * local_cols * datum_size);
*subs = (void **) my_malloc (grid_id,local_rows*PTR_SIZE);
lptr = (void *) *subs;
rptr = (void *) *storage;
for (i = 0; i < local_rows; i++) {
   *(lptr++) = (void *) rptr;
   rptr += local_cols * datum_size;
}

/* Grid process 0 reads in the matrix one row at a time
   and distributes each row among the MPI processes. */

if (grid_id == 0)
   buffer = my_malloc (grid_id, *n * datum_size);
```

```
/* For each row of processes in the process grid... */
for (i = 0; i < grid_size[0]; i++) {
   coords[0] = i;

   /* For each matrix row controlled by this proc row...*/
   for (j = 0; j < BLOCK_SIZE(i,grid_size[0],*m); j++) {

      /* Read in a row of the matrix */

      if (grid_id == 0) {
         fread (buffer, datum_size, *n, infileptr);
      }

      /* Distribute it among processes in the grid row */

      for (k = 0; k < grid_size[1]; k++) {
         coords[1] = k;

         /* Find address of first element to send */
         raddr = buffer +
            BLOCK_LOW(k,grid_size[1],*n) * datum_size;

         /* Determine the grid ID of the process getting
            the subrow */

         MPI_Cart_rank (grid_comm, coords, &dest_id);

         /* Process 0 is responsible for sending...*/
         if (grid_id == 0) {

            /* It is sending (copying) to itself */
            if (dest_id == 0) {
               laddr = (*subs)[j];
               memcpy (laddr, raddr,
                  local_cols * datum_size);

            /* It is sending to another process */
            } else {
               MPI_Send (raddr,
                  BLOCK_SIZE(k,grid_size[1],*n), dtype,
               dest_id, 0, grid_comm);
            }

         /* Process 'dest_id' is responsible for
            receiving... */
         } else if (grid_id == dest_id) {
            MPI_Recv ((*subs)[j], local_cols, dtype, 0,
               0, grid_comm,&status);
         }
```

```
        }
      }
   }
   if (grid_id == 0) free (buffer);
}


/*
 *   Function 'read_col_striped_matrix' reads a matrix from a
 *   file.  The first two elements of the file are integers
 *   whose values are the dimensions of the matrix ('m' rows
 *   and 'n' columns).  What follows are 'm''n' values
 *   representing the matrix elements stored in row-major
 *   order.  This function allocates blocks of columns of the
 *   matrix to the MPI processes.
 */


void read_col_striped_matrix (
      char         *s,        /* IN - File name */
      void       ***subs,     /* OUT - 2-D array */
      void        **storage,  /* OUT - Array elements */
      MPI_Datatype dtype,     /* IN - Element type */
      int          *m,        /* OUT - Rows */
      int          *n,        /* OUT - Cols */
      MPI_Comm      comm)     /* IN - Communicator */
{
   void      *buffer;         /* File buffer */
   int        datum_size;     /* Size of matrix element */
   int        i, j;
   int        id;             /* Process rank */
   FILE      *infileptr;      /* Input file ptr */
   int        local_cols;     /* Cols on this process */
   void     **lptr;           /* Pointer into 'subs' */
   void      *rptr;           /* Pointer into 'storage' */
   int        p;              /* Number of processes */
   int       *send_count;     /* Each proc's count */
   int       *send_disp;      /* Each proc's displacement */

   MPI_Comm_size (comm, &p);
   MPI_Comm_rank (comm, &id);
   datum_size = get_size (dtype);

   /* Process p-1 opens file, gets number of rows and
      cols, and broadcasts this info to other procs. */

   if (id == (p-1)) {
      infileptr = fopen (s, "r");
      if (infileptr == NULL) *m = 0;
```

```
      else {
         fread (m, sizeof(int), 1, infileptr);
         fread (n, sizeof(int), 1, infileptr);
      }
   }
   MPI_Bcast (m, 1, MPI_INT, p-1, comm);

   if (!(*m)) MPI_Abort (comm, OPEN_FILE_ERROR);

   MPI_Bcast (n, 1, MPI_INT, p-1, comm);

   local_cols = BLOCK_SIZE(id,p,*n);

   /* Dynamically allocate two-dimensional matrix 'subs' */

   *storage = my_malloc (id, *m * local_cols * datum_size);
   *subs = (void **) my_malloc (id, *m * PTR_SIZE);
   lptr = (void *) *subs;

   rptr = (void *) *storage;
   for (i = 0; i < *m; i++) {
      *(lptr++) = (void *) rptr;
      rptr += local_cols * datum_size;
   }

   /* Process p-1 reads in the matrix one row at a time and
      distributes each row among the MPI processes. */

   if (id == (p-1))
      buffer = my_malloc (id, *n * datum_size);
   create_mixed_xfer_arrays (id,p,*n,&send_count,&send_disp);
   for (i = 0; i < *m; i++) {
      if (id == (p-1))
         fread (buffer, datum_size, *n, infileptr);
      MPI_Scatterv (buffer, send_count, send_disp, dtype,
         (*storage)+i*local_cols*datum_size, local_cols,
         dtype, p-1, comm);
   }
   free (send_count);
   free (send_disp);
   if (id == (p-1)) free (buffer);
}


/*
 *   Process p-1 opens a file and inputs a two-dimensional
 *   matrix, reading and distributing blocks of rows to the
 *   other processes.
 */
```

```
void read_row_striped_matrix (
   char        *s,          /* IN - File name */
   void      ***subs,       /* OUT - 2D submatrix indices */
   void       **storage,    /* OUT - Submatrix stored here */
   MPI_Datatype dtype,      /* IN - Matrix element type */
   int         *m,          /* OUT - Matrix rows */
   int         *n,          /* OUT - Matrix cols */
   MPI_Comm     comm)       /* IN - Communicator */
{
   int          datum_size;   /* Size of matrix element */
   int          i;
   int          id;           /* Process rank */
   FILE        *infileptr;    /* Input file pointer */
   int          local_rows;   /* Rows on this proc */
   void       **lptr;         /* Pointer into 'subs' */
   int          p;            /* Number of processes */
   void        *rptr;         /* Pointer into 'storage' */
   MPI_Status   status;       /* Result of receive */
   int          x;            /* Result of read */

   MPI_Comm_size (comm, &p);
   MPI_Comm_rank (comm, &id);
   datum_size = get_size (dtype);

   /* Process p-1 opens file, reads size of matrix,
      and broadcasts matrix dimensions to other procs */

   if (id == (p-1)) {
      infileptr = fopen (s, "r");
      if (infileptr == NULL) *m = 0;
      else {
         fread (m, sizeof(int), 1, infileptr);
         fread (n, sizeof(int), 1, infileptr);
      }
   }
   MPI_Bcast (m, 1, MPI_INT, p-1, comm);

   if (!(*m)) MPI_Abort (MPI_COMM_WORLD, OPEN_FILE_ERROR);

   MPI_Bcast (n, 1, MPI_INT, p-1, comm);

   local_rows = BLOCK_SIZE(id,p,*m);

   /* Dynamically allocate matrix. Allow double subscripting
      through 'a'. */

   *storage = (void *) my_malloc (id,
       local_rows * *n * datum_size);
   *subs = (void **) my_malloc (id, local_rows * PTR_SIZE);
```

```
   lptr = (void *) &(*subs[0]);
   rptr = (void *) *storage;
   for (i = 0; i < local_rows; i++) {
      *(lptr++)= (void *) rptr;
      rptr += *n * datum_size;
   }

   /* Process p-1 reads blocks of rows from file and
      sends each block to the correct destination process.
      The last block it keeps. */

   if (id == (p-1)) {
      for (i = 0; i < p-1; i++) {
         x = fread (*storage, datum_size,
            BLOCK_SIZE(i,p,*m) * *n, infileptr);
         MPI_Send (*storage, BLOCK_SIZE(i,p,*m) * *n, dtype,
            i, DATA_MSG, comm);
      }
      x = fread (*storage, datum_size, local_rows * *n,
         infileptr);
      fclose (infileptr);
   } else
      MPI_Recv (*storage, local_rows * *n, dtype, p-1,
         DATA_MSG, comm, &status);
}


/*
 *   Open a file containing a vector, read its contents,
 *   and distribute the elements by block among the
 *   processes in a communicator.
 */

void read_block_vector (
    char        *s,       /* IN - File name */
    void       **v,       /* OUT - Subvector */
    MPI_Datatype dtype,   /* IN - Element type */
    int         *n,       /* OUT - Vector length */
    MPI_Comm     comm)    /* IN - Communicator */
{
   int         datum_size;   /* Bytes per element */
   int         i;
   FILE       *infileptr;    /* Input file pointer */
   int         local_els;    /* Elements on this proc */
   MPI_Status  status;       /* Result of receive */
   int         id;           /* Process rank */
   int         p;            /* Number of processes */
   int         x;            /* Result of read */
```

```
   datum_size = get_size (dtype);
   MPI_Comm_size(comm, &p);
   MPI_Comm_rank(comm, &id);

   /* Process p-1 opens file, determines number of vector
      elements, and broadcasts this value to the other
      processes. */

   if (id == (p-1)) {
      infileptr = fopen (s, "r");
      if (infileptr == NULL) *n = 0;
      else fread (n, sizeof(int), 1, infileptr);
   }
   MPI_Bcast (n, 1, MPI_INT, p-1, comm);
   if (! *n) {
      if (!id) {
         printf ("Input file '%s' cannot be opened\n", s);
         fflush (stdout);
      }
   }

   /* Block mapping of vector elements to processes */

   local_els = BLOCK_SIZE(id,p,*n);

   /* Dynamically allocate vector. */

   *v = my_malloc (id, local_els * datum_size);
   if (id == (p-1)) {
      for (i = 0; i < p-1; i++) {
         x = fread (*v, datum_size, BLOCK_SIZE(i,p,*n),
            infileptr);
         MPI_Send (*v, BLOCK_SIZE(i,p,*n), dtype, i, DATA_MSG,
            comm);
      }
      x = fread (*v, datum_size, BLOCK_SIZE(id,p,*n),
            infileptr);
      fclose (infileptr);
   } else {
      MPI_Recv (*v, BLOCK_SIZE(id,p,*n), dtype, p-1, DATA_MSG,
         comm, &status);
   }
}


/*   Open a file containing a vector, read its contents,
     and replicate the vector among all processes in a
     communicator. */
```

```
void read_replicated_vector (
   char         *s,       /* IN - File name */
   void         **v,      /* OUT - Vector */
   MPI_Datatype dtype,    /* IN - Vector type */
   int          *n,       /* OUT - Vector length */
   MPI_Comm     comm)     /* IN - Communicator */
{
   int         datum_size; /* Bytes per vector element */
   int         i;
   int         id;         /* Process rank */
   FILE        *infileptr; /* Input file pointer */
   int         p;          /* Number of processes */

   MPI_Comm_rank (comm, &id);
   MPI_Comm_size (comm, &p);
   datum_size = get_size (dtype);
   if (id == (p-1)) {
      infileptr = fopen (s, "r");
      if (infileptr == NULL) *n = 0;
      else fread (n, sizeof(int), 1, infileptr);
   }
   MPI_Bcast (n, 1, MPI_INT, p-1, MPI_COMM_WORLD);
   if (! *n) terminate (id, "Cannot open vector file");

   *v = my_malloc (id, *n * datum_size);

   if (id == (p-1)) {
      fread (*v, datum_size, *n, infileptr);
      fclose (infileptr);
   }
   MPI_Bcast (*v, *n, dtype, p-1, MPI_COMM_WORLD);
}

/******************** OUTPUT FUNCTIONS ********************/

/*
 *   Print elements of a doubly subscripted array.
 */

void print_submatrix (
   void         **a,      /* OUT - Doubly subscripted array */
   MPI_Datatype dtype,    /* OUT - Type of array elements */
   int          rows,     /* OUT - Matrix rows */
   int          cols)     /* OUT - Matrix cols */
{
   int i, j;

   for (i = 0; i < rows; i++) {
      for (j = 0; j < cols; j++) {
```

```
         if (dtype == MPI_DOUBLE)
            printf ("%6.3f ", ((double **)a)[i][j]);
         else {
            if (dtype == MPI_FLOAT)
               printf ("%6.3f ", ((float **)a)[i][j]);
            else if (dtype == MPI_INT)
               printf ("%6d ", ((int **)a)[i][j]);
         }
      }
      putchar ('\n');
   }
}


/*
 *   Print elements of a singly subscripted array.
 */

void print_subvector (
   void        *a,        /* IN - Array pointer */
   MPI_Datatype dtype,    /* IN - Array type */
   int          n)        /* IN - Array size */
{
   int i;

   for (i = 0; i < n; i++) {
      if (dtype == MPI_DOUBLE)
         printf ("%6.3f ", ((double *)a)[i]);
      else {
         if (dtype == MPI_FLOAT)
            printf ("%6.3f ", ((float *)a)[i]);
         else if (dtype == MPI_INT)
            printf ("%6d ", ((int *)a)[i]);
      }
   }
}


/*
 *   Print a matrix distributed checkerboard fashion among
 *   the processes in a communicator.
 */

void print_checkerboard_matrix (
   void       **a,             /* IN -2D matrix */
   MPI_Datatype dtype,         /* IN -Matrix element type */
   int          m,             /* IN -Matrix rows */
   int          n,             /* IN -Matrix columns */
   MPI_Comm     grid_comm)     /* IN - Communicator */
```

```
{
   void      *buffer;          /* Room to hold 1 matrix row */
   int        coords[2];       /* Grid coords of process
                                  sending elements */
   int        datum_size;      /* Bytes per matrix element */
   int        els;             /* Elements received */
   int        grid_coords[2]; /* Coords of this process */
   int        grid_id;         /* Process rank in grid */
   int        grid_period[2]; /* Wraparound */
   int        grid_size[2];    /* Dims of process grid */
   int        i, j, k;
   void      *laddr;           /* Where to put subrow */
   int        local_cols;      /* Matrix cols on this proc */
   int        p;               /* Number of processes */
   int        src;             /* ID of proc with subrow */
   MPI_Status status;          /* Result of receive */

   MPI_Comm_rank (grid_comm, &grid_id);
   MPI_Comm_size (grid_comm, &p);
   datum_size = get_size (dtype);

   MPI_Cart_get (grid_comm, 2, grid_size, grid_period,
      grid_coords);
   local_cols = BLOCK_SIZE(grid_coords[1],grid_size[1],n);

   if (!grid_id)
      buffer = my_malloc (grid_id, n*datum_size);

   /* For each row of the process grid */
   for (i = 0; i < grid_size[0]; i++) {
      coords[0] = i;

      /* For each matrix row controlled by the process row */
      for (j = 0; j < BLOCK_SIZE(i,grid_size[0],m); j++) {

         /* Collect the matrix row on grid process 0 and
            print it. */
         if (!grid_id) {
            for (k = 0; k < grid_size[1]; k++) {
               coords[1] = k;
               MPI_Cart_rank (grid_comm, coords, &src);
               els = BLOCK_SIZE(k,grid_size[1],n);
               laddr = buffer +
                  BLOCK_LOW(k,grid_size[1],n) * datum_size;
               if (src == 0) {
                  memcpy (laddr, a[j], els * datum_size);
               } else {
                  MPI_Recv(laddr, els, dtype, src, 0,
                     grid_comm, &status);
```

```
            }
         }
         print_subvector (buffer, dtype, n);
         putchar ('\n');
      } else if (grid_coords[0] == i) {
         MPI_Send (a[j], local_cols, dtype, 0, 0,
            grid_comm);
      }
   }
}
if (!grid_id) {
   free (buffer);
   putchar ('\n');
}
}

/*
 *   Print a matrix that has a columnwise-block-striped data
 *   decomposition among the elements of a communicator.
 */

void print_col_striped_matrix (
   void        **a,        /* IN - 2D array */
   MPI_Datatype dtype,     /* IN - Type of matrix elements */
   int          m,         /* IN - Matrix rows */
   int          n,         /* IN - Matrix cols */
   MPI_Comm     comm)      /* IN - Communicator */
{
   MPI_Status status;      /* Result of receive */
   int        datum_size; /* Bytes per matrix element */
   void      *buffer;      /* Enough room to hold 1 row */
   int        i, j;
   int        id;          /* Process rank */
   int        p;           /* Number of processes */
   int*       rec_count;   /* Elements received per proc */
   int*       rec_disp;    /* Offset of each proc's block */

   MPI_Comm_rank (comm, &id);
   MPI_Comm_size (comm, &p);
   datum_size = get_size (dtype);
   create_mixed_xfer_arrays (id, p, n, &rec_count,&rec_disp);

   if (!id)
      buffer = my_malloc (id, n*datum_size);

   for (i = 0; i < m; i++) {
      MPI_Gatherv (a[i], BLOCK_SIZE(id,p,n), dtype, buffer,
         rec_count, rec_disp, dtype, 0, MPI_COMM_WORLD);
```

```
      if (!id) {
         print_subvector (buffer, dtype, n);
         putchar ('\n');
      }
   }
   free (rec_count);
   free (rec_disp);
   if (!id) {
      free (buffer);
      putchar ('\n');
   }
}


/*
 *   Print a matrix that is distributed in row-striped
 *   fashion among the processes in a communicator.
 */

void print_row_striped_matrix (
   void **a,              /* IN - 2D array */
   MPI_Datatype dtype,    /* IN - Matrix element type */
   int m,                 /* IN - Matrix rows */
   int n,                 /* IN - Matrix cols */
   MPI_Comm comm)         /* IN - Communicator */
{
   MPI_Status  status;          /* Result of receive */
   void       *bstorage;        /* Elements received from
                                   another process */
   void      **b;               /* 2D array indexing into
                                   'bstorage' */
   int         datum_size;      /* Bytes per element */
   int         i;
   int         id;              /* Process rank */
   int         local_rows;      /* This proc's rows */
   int         max_block_size;  /* Most matrix rows held by
                                   any process */
   int         prompt;          /* Dummy variable */
   int         p;               /* Number of processes */

   MPI_Comm_rank (comm, &id);
   MPI_Comm_size (comm, &p);
   local_rows = BLOCK_SIZE(id,p,m);
   if (!id) {
      print_submatrix (a, dtype, local_rows, n);
      if (p > 1) {
         datum_size = get_size (dtype);
         max_block_size = BLOCK_SIZE(p-1,p,m);
         bstorage = my_malloc (id,
```

```
            max_block_size * n * datum_size);
         b = (void **) my_malloc (id,
            max_block_size * datum_size);
         b[0] = bstorage;
         for (i = 1; i < max_block_size; i++) {
            b[i] = b[i-1] + n * datum_size;
         }
         for (i = 1; i < p; i++) {
            MPI_Send (&prompt, 1, MPI_INT, i, PROMPT_MSG,
               MPI_COMM_WORLD);
            MPI_Recv (bstorage, BLOCK_SIZE(i,p,m)*n, dtype,
               i, RESPONSE_MSG, MPI_COMM_WORLD, &status);
            print_submatrix (b, dtype, BLOCK_SIZE(i,p,m), n);
         }
         free (b);
         free (bstorage);
      }
      putchar ('\n');
   } else {
      MPI_Recv (&prompt, 1, MPI_INT, 0, PROMPT_MSG,
         MPI_COMM_WORLD, &status);
      MPI_Send (*a, local_rows * n, dtype, 0, RESPONSE_MSG,
         MPI_COMM_WORLD);
   }
}


/*
 *   Print a vector that is block distributed among the
 *   processes in a communicator.
 */

void print_block_vector (
   void        *v,       /* IN - Address of vector */
   MPI_Datatype dtype,   /* IN - Vector element type */
   int          n,       /* IN - Elements in vector */
   MPI_Comm     comm)    /* IN - Communicator */
{
   int        datum_size; /* Bytes per vector element */
   int        i;
   int        prompt;     /* Dummy variable */
   MPI_Status status;     /* Result of receive */
   void       *tmp;       /* Other process's subvector */
   int        id;         /* Process rank */
   int        p;          /* Number of processes */

   MPI_Comm_size (comm, &p);
   MPI_Comm_rank (comm, &id);
   datum_size = get_size (dtype);
```

```
   if (!id) {
      print_subvector (v, dtype, BLOCK_SIZE(id,p,n));
      if (p > 1) {
         tmp = my_malloc (id,BLOCK_SIZE(p-1,p,n)*datum_size);
         for (i = 1; i < p; i++) {
            MPI_Send (&prompt, 1, MPI_INT, i, PROMPT_MSG,
               comm);
            MPI_Recv (tmp, BLOCK_SIZE(i,p,n), dtype, i,
               RESPONSE_MSG, comm, &status);
            print_subvector (tmp, dtype, BLOCK_SIZE(i,p,n));
         }
         free (tmp);
      }
      printf ("\n\n");
   } else {
      MPI_Recv (&prompt, 1, MPI_INT, 0, PROMPT_MSG, comm,
         &status);
      MPI_Send (v, BLOCK_SIZE(id,p,n), dtype, 0,
         RESPONSE_MSG, comm);
   }
}


/*
 *   Print a vector that is replicated among the processes
 *   in a communicator.
 */

void print_replicated_vector (
   void        *v,      /* IN - Address of vector */
   MPI_Datatype dtype,  /* IN - Vector element type */
   int          n,      /* IN - Elements in vector */
   MPI_Comm     comm)   /* IN - Communicator */
{
   int id;              /* Process rank */

   MPI_Comm_rank (comm, &id);

   if (!id) {
      print_subvector (v, dtype, n);
      printf ("\n\n");
   }
}
```

APPENDIX C

Debugging MPI Programs

C.1 INTRODUCTION

Programming is an error-prone activity. While careful design is perhaps the single most important step in developing a correct program, virtually every programmer writes programs that need debugging. Most programmers use a symbolic debugger to isolate some bugs and `printf` statements to find the rest; the amount of time spent with each methodology depends upon the programmer's skill and the complexity of the application.

Debugging parallel programs is much harder than debugging serial programs. First, there is much more to go wrong. Multiple processes are performing computations that interact with each other through a variety of message-passing functions. Second, parallel debuggers are not as advanced as serial debuggers. Typically, parallel programmers do not have access to good tools.

This appendix lists the kinds of bugs typically found in MPI programs, and it provides some rules of thumb for debugging your programs.

C.2 TYPICAL BUGS IN MPI PROGRAMS

C.2.1 Bugs Leading to Deadlock

A process is **deadlocked** if it is "blocked waiting for a condition that will never become true" [3]. An MPI program will not be completed if one or more of its processes are deadlocked. You can often trace deadlock in an MPI program to one of the two following bugs:

Deadlock Bug 1 A single process calls a collective communication function. For example, only the root process calls `MPI_Reduce` or `MPI_Bcast`.

Prevention Do not put a call to a collective communication function inside conditionally executed code. If you must put the call inside conditionally executed

code, ensure that the conditional expression evaluates to the same value on every process, so that either all or none of the processes enter the block of code containing the collective communication function.

Deadlock Bug 2 Two or more processes are trying to exchange data, but all call a blocking receive function such as `MPI_Recv` before any calls an MPI send function.

Prevention There are several ways to prevent this bug. First, you could structure your program so that processes always call `MPI_Send` or another message-sending function before they call `MPI_Recv`. Second, you could replace the pair of function calls `MPI_Send` and `MPI_Recv` with the single function call `MPI_Sendrecv`, which is guaranteed not to deadlock. Third, you could replace the blocking call `MPI_Recv` with the nonblocking call `MPI_Irecv` and place the matching call to `MPI_Wait` after the call to `MPI_Send`.

Deadlock Bug 3 A process tries to receive data from a process that will never send it, resulting in deadlock.

Prevention If the rank number does not correspond to a process in the communicator, the MPI run-time system will catch the error. However, this does not help you if the rank is in the acceptable range. The best way to avoid this bug is to use collective communications functions whenever possible. If point-to-point communications are necessary, keep the communication pattern simple.

Deadlock Bug 4 A process tries to receive data from itself.

Prevention A simple examination of the source code can weed out instances of this bug. As an alternative, you could put a run-time check before each call to a receive function.

C.2.2 Bugs Leading to Incorrect Results

Incorrect Result Bug 1 Type mismatch between send and receive. For example, the sending process may put a message with element type `MPI_INT` into the buffer, while the receiving process may read a message with element type `MPI_FLOAT`.

Prevention Structure your program so that each call to a message-sending function has exactly one matching call to a message-receiving function. Make sure it is easy to determine the receive that goes with each send. Double-check to ensure that the paired function calls assume the message has the same length and the same element type.

Incorrect Result Bug 2 Mixed-up parameters. An example of this kind of error is reversing the order of the first and second parameters to `MPI_Reduce`.

Prevention Most MPI functions have many parameters. Your safest strategy is to refer to the function headers in Appendix A whenever coding up an MPI function, to ensure that you put the arguments in the correct order.

C.2.3 Advantages of Collective Communications

There are more opportunities for bugs with point-to-point communications (e.g., send and receive) than with collective communications (e.g., broadcast and reduce). In a collective communication, typically all processes are at the same point of execution in the program. All processes usually invoke the function from the same line of the source program. Hence all the arguments are the same. If one process has called the function correctly, all have.

In contrast, consider point-to-point communications. In most local communications the sender and the receiver are calling different MPI functions, creating opportunities for argument mismatches or other errors. It is possible to indicate the wrong source or destination, to get the type of one or more arguments wrong, to get wrong the number of data elements being passed, or to specify the wrong tag.

For these reasons, use collective communications whenever it makes sense.

C.3 PRACTICAL DEBUGGING STRATEGIES

- If the parallel program will run on a single process, your first step should be to get the one-process version of the parallel program working correctly. It often tests much of the program's functionality, such as I/O. More importantly, you can take advantage of a sequential debugger to set breakpoints, test values, etc.
- After you have the program working correctly with one process, work with the smallest number of processes that allows all of the program's functionality to be exercised. Usually two or three processes are sufficient.
- Work with the smallest problem size that exercises all of the program's functionality. For example, when writing a program that solves a system of linear equations, a 4×4 system might exercise the same functionality as a 1024×1024 system. By using a smaller problem size, you can put in `printf` statements that let you look at entire data structures. In addition, since the program has less output, the output you do get will be easier to understand.
- Put `fflush(stdout);` after every `printf` statement. Otherwise, you may not get all of the output produced by every process before your program crashes/deadlocks.
- For point-to-point messages, print the data that are being sent and print the data that are received to make sure that the values match.
- The messages received from any one process will be in chronological order, but messages received from different processes do not necessarily arrive in chronological order. Do not assume if a message from process X appears before a message from process Y, that message X was printed before message Y. Prefix each message with the process rank, then run the output

of the program through the Unix `sort` utility. This will organize output by process, which is about as well as you can do.

- First debug the initialization phase of the program to make sure that all the data structures have been set up correctly.
- Check to make sure that the local indices used to access local data on each processor are correctly calculated.
- When debugging the program, do not combine messages or use complex data structures to optimize performance. First get the logic right. Then worry about combining messages or taking other performance-enhancing steps.

Review of Complex Numbers

This appendix reviews how to perform arithmetic on complex numbers. The material closely follows the presentation of Weaver [112].

A *complex number* is an ordered pair of real numbers, denoted (x, y). We call x the *real part* of the complex number and y the *imaginary part*. Two complex numbers (x_1, y_1) and (x_2, y_2) are equal if and only if $x_1 = x_2$ and $y_1 = y_2$.

Let $z_1 = (x_1, y_1)$ and $z_2 = (x_2, y_2)$ be two complex numbers. The sum of these complex numbers is

$$z_1 + z_2 = (x_1 + x_2, y_1 + y_2)$$

The product of these complex numbers is

$$z_1 z_2 = (x_1 x_2 - y_1 y_2, x_1 y_2 + y_1 x_2)$$

Addition and multiplication with complex numbers is commutative, associative, and distributive.

Any real number x can be represented as the complex number $(x, 0)$.

Three special complex numbers are the zero element, the unit element, and the imaginary unit element.

The *zero element*, denoted $\mathbf{0}$, is the complex number $(0, 0)$.

The sum of any complex number z and the zero element is z:

$$z + \mathbf{0} = (x, y) + (0, 0) = (x + 0, y + 0) = (x, y) = z$$

The product of any complex number z and the zero element is $\mathbf{0}$:

$$z\mathbf{0} = (x, y)(0, 0) = (x \times 0 - y \times 0, x \times 0 + y \times 0) = (0, 0) = \mathbf{0}$$

The *unit element*, denoted $\mathbf{1}$, is the complex number $(1, 0)$.

The product of any complex number z and the unit element is z:

$$z \times \mathbf{1} = (x, y)(1, 0) = (x \times 1 - y \times 0, 1 \times y + 0 \times x) = (x, y) = z$$

The *imaginary element*, denoted i, is the complex number $(0, 1)$. The imaginary element is the square root of -1:

$$i^2 = (0, 1)(0, 1) = (0 \times 0 - 1 \times 1, 0 \times 1 + 1 \times 0) = (-1, 0) = -1$$

Theorem D.1

Every complex number $z = (x, y)$ can be represented as $x + iy$.

Proof

The real number $x = (x, 0)$. The product of real number y with the imaginary element i is

$$iy = (0, 1)(y, 0) = (0 \times y - 1 \times 0, 0 \times 0 + y \times 1) = (0, y)$$

Hence

$$x + iy = (x, 0) + (0, y) = (x, y)$$

See Figure D.1. We have represented the complex number z as $x + iy$, where the horizontal axis corresponds to the real part of z and the vertical axis corresponds to the imaginary part of z.

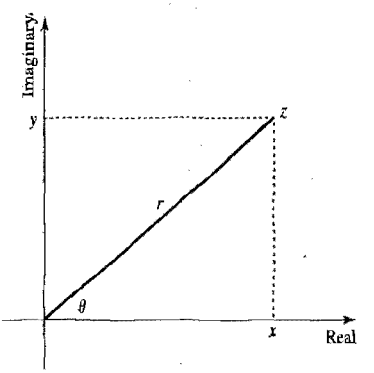


Figure D.1 Every complex number z can be represented as an ordered pair of real numbers (x, y), where x is the real part and y is the imaginary part. It can also be represented as a vector having length r and angle θ, where θ is measured counterclockwise from the real axis.

We can also think of z as a vector having length r and angle θ, where θ is measured counterclockwise from the real axis. Note that

$$x = r\cos\theta$$
$$y = r\sin\theta$$

Using these equations we can write $z = x + iy = r(\cos\theta + i\sin\theta)$.

When we study the discrete Fourier transform we want to represent z in exponential form, which we derive here. Using Taylor's series we can show

$$\sin\theta = \theta - \frac{\theta^3}{3} + \frac{\theta^5}{5} - \frac{\theta^7}{7} + \cdots$$

$$\cos\theta = 1 - \frac{\theta^2}{2} + \frac{\theta^4}{4} - \frac{\theta^6}{6} + \cdots$$

$$e^{i\theta} = 1 + i\theta - \frac{\theta^2}{2} - \frac{i\theta^3}{3} + \frac{\theta^4}{4} + \frac{i\theta^5}{5} + \cdots$$

$$= \left(1 - \frac{\theta^2}{2} + \frac{\theta^4}{4} + \cdots\right) + i\left(\theta - \frac{\theta^3}{3} + \frac{\theta^5}{5} + \cdots\right)$$

Combining these equations yields

$$e^{i\theta} = \cos\theta + i\sin\theta$$

and

$$e^{-i\theta} = \cos\theta - i\sin\theta$$

Recall that $z = x + iy = r(\cos\theta + i\sin\theta)$. Hence

$$z = re^{i\theta}$$

is another way to represent a complex number.

One property of the exponential representation of complex numbers is that it simplifies multiplication and division. Let $z_1 = r_1e^{\theta_1}$ and $z_2 = r_2e^{\theta_2}$ be two complex numbers. Then

$$z_1z_2 = r_1e^{\theta_1}r_2e^{\theta_2} = r_1r_2e^{\theta_1+\theta_2}$$
$$z_1/z_2 = (r_1e^{\theta_1})/(r_2e^{\theta_2}) = (r_1/r_2)e^{\theta_1-\theta_2}$$

A *complex nth root of unity* is a complex number ω such that $\omega^n = 1$, the unit element.

There are exactly n complex nth roots of unity, represented by $e^{2\pi ik/n}$ for $k = 1, 2, \ldots, n$.

The complex number $e^{2\pi i/n}$, denoted ω_n, is the *principal nth root of unity*.

Figure D.2 illustrates the principal eighth root of unity and its powers, the other complex eighth roots of unity.

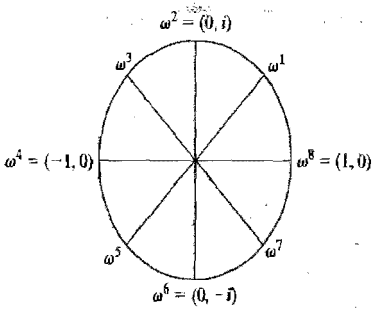


Figure D.2 The principal eighth root of unity and its powers.

Lemma D.1

(Cancellation lemma) For any integers $n > 0$, $k > 0$, and $d > 0$, $\omega_{dn}^{dk} = \omega_n^k$.

Proof

$$\omega_{dn}^{dk} = (e^{2\pi i/(dn)})^{dk} = (e^{2\pi i/n})^k = \omega_n^k$$

Corollary D.1

For any even integer $n > 0$, $\omega_n^{n/2} = \omega_2 = -1$.

Proof

$$\omega_n^{n/2} = \omega_{2(n/2)}^{n/2} = \omega_2^1 = \omega_2 = -1$$

Lemma D.2

(Halving lemma) If n is an even positive number, then the squares of the n complex nth roots of unity are identical to the $n/2$ complex $(n/2)$th roots of unity.

Proof

By the cancellation lemma we know that if k is nonnegative then $(\omega_n^k)^2 = \omega_{n/2}^k$. If we square all of the complex nth roots of unity, we get every $(n/2)$th root of unity twice, because

$$(\omega_n^{k+n/2})^2 = \omega_n^{2k+n} = \omega_n^{2k}\omega_n^n = \omega_n^{2k} = (\omega_n^k)^2$$

APPENDIX E

OpenMP Functions

This appendix describes all the C/C++ functions in the OpenMP standard. Every function has either no parameters or one parameter. All parameters are input parameters, where the caller provides the value. All results are returned to the user through the function's return value.

`int omp_get_dynamic (void)`

Function `omp_get_dynamic` returns 1 if dynamic threads are enabled and 0 if they are disabled.

`int omp_get_max_threads (void)`

Function `omp_get_max_threads` returns an integer whose value is the maximum number of threads that the run-time system will let your program create.

`int omp_get_nested (void)`

Function `omp_get_nested` returns 1 if nested parallelism is enabled and 0 otherwise. All current OpenMP implementations have nested parallelism disabled by default.

`int omp_get_num_procs (void)`

Function `omp_get_num_procs` returns the number of processors the parallel program can use.

`int omp_get_num_threads (void)`

Function `omp_get_num_threads` returns the number of threads that are currently active. If it is called from a serial portion of the program, the function returns 1.

```
int omp_get_thread_num (void)
```

Function `omp_get_thread_num` returns the thread's identification number. If there are t active threads, the thread identification numbers range from 0 to $t-1$.

```
int omp_in_parallel (void)
```

Function `omp_in_parallel` returns 1 if it has been called inside a parallel block and 0 otherwise.

```
void omp_set_dynamic (
   int k   /* 1 = ON, 0 = FALSE */)
```

Function `omp_set_dynamic` can be used to enable or disable dynamic threads. If dynamic threads are enabled, the run-time system may adjust the number of active threads to match the number of physical processors available. You may wish to disable dynamic threads if you want to know exactly how many threads are created when parallel regions are entered.

```
void omp_set_nested (
   int k      /* 1 = enable; 0 = disable */
)
```

Function `omp_set_nested` is used to enable or disable nested parallelism. Current implementations of OpenMP only support one level of parallelism. Nested parallelism is turned off by default, and activating it has no effect. Hence this function call has no value in current OpenMP implementations.

```
void omp_set_num_threads (
   int t              /* Number of threads desired */
)
```

Function `omp_set_num_threads` sets the desired number of parallel threads for subsequent executions of parallel regions. The number of threads may exceed the number of available processors, in which case multiple threads may be mapped to the same processor. This call must be made from a serial portion of a program.

BIBLIOGRAPHY

1. Akl, S. G. *Parallel Sorting Algorithms.* Orlando, FL: Academic Press, 1985.
2. Amdahl, G. "Validity of the single processor approach to achieving large scale computing capabilities." In *AFIPS Conference Proceedings,* Vol. 30, pages 483–485, Washington, D.C.: Thompson Books, April 1967.
3. Andrews, Gregory R. *Concurrent Programming: Principles and Practice.* Redwood City, CA: Benjamin/Cummings, 1991.
4. Anton, H. *Elementary Linear Algebra.* 3d ed. New York: John Wiley & Sons, 1981.
5. Baase, Sara, and Allen Van Gelder. *Computer Algorithms: Introduction to Design and Analysis.* 3d ed. Reading, MA: Addison-Wesley, 2000.
6. Babb, Robert G. II. Introduction. In Robert G. Babb II, (ed.), *Programming Parallel Processors,* pages 1–6. Reading, MA: Addison-Wesley, 1988.
7. Bacon, David F., Susan L. Graham, and Oliver J. Sharp. "Compiler transformations for high-performance computing." *ACM Computing Surveys* 26(4):345–420, December 1994.
8. Batcher, K. E. "Sorting networks and their applications." In *Proceedings of the AFIPS Spring Joint Computer Conference,* Vol. 32, pages 307–314. Reston, VA: AFIPS Press, 1968.
9. Bertsekas, D. P., and J. N. Tsitsiklis. *Parallel and Distributed Computation: Numerical Methods.* Englewood Cliffs, NJ: Prentice-Hall, 1989.
10. Bhattacharya, S., and A. Bagchi. "Searching game trees in parallel using SSS*." In *Proceedings AAAI-90,* pages 42–47. The AAAI Press, 1990.
11. Bressoud, David M. *Factorization and Primality Testing.* New York: Springer-Verlag, 1989.
12. Browne, James C., Syed I. Hyder, Jack Dongarra, Keith Moore, and Peter Newton. "Visual programming and debugging for parallel computing." *IEEE Parallel & Distributed Technology* 3(1):75–83, Spring 1995.
13. Camp, W. J., S. J. Plimpton, B. A. Hendrickson, and R. W. Leland. "Massively parallel methods for engineering and science problems." *Communications of the ACM* 37(4):30–41, April 1994.
14. Cannon, L. E. *A Cellular Computer to Implement the Kalman Filter Algorithm.* PhD thesis, Montana State University, Bozeman, MT, 1969.
15. Carriero, Nicholas, and David Gelernter. *How to Write Parallel Programs: A First Course.* Cambridge, MA: The MIT Press, 1990.
16. Chandra, Rohit, Leonardo Dagum, Dave Kohr, Dror Maydan, Jeff McDonald, and Ramesh Menon. *Parallel Programming in OpenMP.* San Francisco: Morgan Kaufmann, 2001.
17. Coddington, Paul D. "Random number generators for parallel computers." Technical report, Northeast Parallel Architectures Center, Syracuse University, Syracuse, NY, April 1997.
18. Cormen, Thomas H., Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms.* 2d ed. Cambridge, MA: The MIT Press, 2001.
19. de Groot, A. D. *Thought and Choice in Chess.* The Hague: Mouton, 1965.
20. de Kergommeaux, Jacques Chassin, and Philippe Codognet. "Parallel logical programming systems." *ACM Computing Surveys* 26(3): 295–336, September 1994.
21. Dijkstra, E. W., W. H. Seijen, and A. J. M. V. Gasteren. "Derivation of a termination detection algorithm for a distributed computation." *Information Processing Letters* 16(5):217–219, 1983.
22. Dongarra, J. J., I. S. Duff, D. C. Sorenson, and H. A. van der Vorst. *Solving Linear Systems on Vector and Shared Memory Computers.* Philadelphia: SIAM Press, 1991.
23. Fagan, M. J. *Finite Element Analysis: Theory and Practice.* Singapore: Longman Scientific & Technical, 1992.

24. Feitelson, Dror G., Anat Barat, Gabriel Benhanokh, David Er-El, Yoav Etsion, Avi Kavas, Tomer Klainer, Uri Lublin, and Marc A. Volovic. "The ParPar system: A software MPP." In Rajkumar Buyya (ed.), *High Performance Cluster Computing: Architectures and Systems*. Vol. 1, pages 754–770. Upper Saddle River, NJ: Prentice-Hall PTR, 1999.
25. Felten, E. W., and S. W. Otto. "A highly parallel chess program." In *Proceedings of the International Conference on Fifth Generation Computer Systems 1988*, pages 1001–1009. ICOT, 1988.
26. Ferguson, C., and R. E. Korf. "Distributed tree search and its application to alpha-beta pruning." In *Proceedings AAAI-88*, pages 128–132, 1988.
27. Floyd, R. W. Algorithm 97: Shortest path. *Communications of the ACM* 5(6):345, June 1962.
28. Flynn, Michael J. "Very high-speed computing systems." *Proceedings of the IEEE* 54(12): 1901–1909, December 1966.
29. Flynn, Michael J. "Some computer organizations and their effectiveness." *IEEE Transactions on Computers* C-21(9):948–960, September 1972.
30. Flynn, Michael J., and Kevin W. Rudd. "Parallel architectures." *ACM Computing Surveys* 28(1):67–70, March 1996.
31. Foster, Ian. *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*. Reading, MA: Addison-Wesley, 1995.
32. Foster, Ian, and K. M. Chandy. "Fortran M: A language for modular parallel programming." *Journal of Parallel and Distributed Computing* 25(1), 1995.
33. Fox, G. C., M. A. Johnson, G. A. Syzenga, S. W. Otto, J. K. Salmon, and D. W. Walker. *Solving Problems on Concurrent Processors*. Vol. 1. Englewood Cliffs, NJ: Prentice-Hall, 1988.
34. Fox, Geoffrey C., Roy D. Williams, and Paul C. Messina. *Parallel Computing Works*. San Francisco: Morgan Kaufmann, 1994.
35. Francis, R. S., and I. D. Mathieson. "A benchmark parallel sort for shared memory multiprocessors." *IEEE Transactions on Computers* C-37(12):1619–1626, December 1988.
36. Gallivan, K. A., R. J. Plemmons, and A. H. Sameh. "Parallel algorithms for dense linear algebra computations." *SIAM Review* 32:54–135, March 1990.
37. Galloway, Robert L., W. Andrew Bass, and Christopher E. Hockey. "Task-oriented asymmetric multiprocessing for interactive image-guided surgery." *Parallel Computing* 24(9–10):1323–1343, September 1998.
38. Garey, Michael R., and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. San Francisco: W. H. Freeman, 1979.
39. Geist, Al, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. *PVM: Parallel Virtual Machine: A Users' Guide and Tutorial for Networked Parallel Computing*. Cambridge, MA: The MIT Press, 1994.
40. Gill, S. "Parallel programming." *Computer Journal* 1(1):2–10, April 1958.
41. Golub, Gene, and James M. Ortega. *Scientific Computing: An Introduction with Parallel Computing*. Boston, MA: Academic Press, 1993.
42. Goodman, S. E., and S. T. Hedetniemi. *Introduction to the Design and Analysis of Algorithms*. New York: McGraw-Hill, 1977.
43. Grama, Ananth Y., Anshul Gupta, and Vipin Kumar. "Isoefficiency: Measuring the scalability of parallel algorithms and architectures." *IEEE Parallel & Distributed Technology* 1(3):12–21, August 1993.
44. Grama, Ananth, Anshul Gupta, George Karypis, and Vipin Kumar. *Introduction to Parallel Computing*. 2d ed. Harlow, England: Addison-Wesley, 2003.
45. Gropp, William, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. Cambridge, MA: The MIT Press, 1994.
46. Gustafson, John L. "Reevaluating Amdahl's law." *Communications of the ACM* 31(5):532–533, May 1988.
47. Gustafson, John L., Gary R. Montry, and Robert E. Benner. "Development of parallel methods for a 1024-processor hypercube." *SIAM*

Journal on Scientific and Statistical Computing 9(4):609–638, March 1988.

48. Gustavson, F. G. "Recursion leads to automatic variable blocking for dense linear-algebra algorithms." *IBM Journal of Research and Development* 41(6), 1999.
49. Hatcher, Philip J., and Michael J. Quinn. *Data-Parallel Programming on MIMD Computers.* Cambridge, MA: The MIT Press, 1991.
50. Hillis, W. Daniel, and Guy L. Steele, Jr. "Data parallel algorithms." *Communications of the ACM* 29(12):1170–1183, December 1986.
51. Hoare, C. A. R. "Quicksort." *Computer Journal* 5(1):10–15, 1962.
52. Hockney, R. W., and C. R. Jesshope. *Parallel Computers: Architecture, Programming and Algorithms.* Bristol: Adam Hilger Ltd, 1981.
53. Houstis, E. N., J. R. Rice, S. Weerawarana, A. C. Catlin, P. Papachiou, K.-Y. Wang, and M. Gaitatzes. "PELLPACK: A problem-solving environment for PDE-based applications on multicomputer platforms." *ACM Transactions on Mathematical Software* 24(1):30–73, March 1998.
54. Hsu, Feng Hsiung. *Behind Deep Blue: Building the Computer that Defeated the World Chess Champion.* Princeton, NJ: Princeton University Press, 2002.
55. Huntbach, M. M., and F. W. Burton. "Alpha-beta search on virtual tree machines." *Information Sciences* 44:3–17, 1988.
56. Jacobi, C. G. J. "Uber eine neue auflosungsart der bei der methode der kleinsten quadrate vorkommenden linearen gleichungen." *Astr. Nachr.* 22(523):297–306, 1845.
57. Jain, A. K., M. N. Murty, and P. J. Flynn. "Data clustering: A review." *ACM Computing Surveys* 31(3):264–323, 1999.
58. Kalos, Malvin H., and Paula A. Whitlock. *Monte Carlo Methods, Volume 1: Basics.* New York: John Wiley & Sons, 1986.
59. Karp, Alan H., and Horace P. Flatt. "Measuring parallel processor performance." *Communications of the ACM* 33(5):539–543, May 1990.
60. Kauffmann, William J. III, and Larry L. Smarr. *Supercomputing and the Transformation of Science.* New York: Scientific American Library, 1993.
61. Kernighan, Brian W., and Dennis M. Ritchie. *The C Programming Language.* Englewood Cliffs, NJ: Prentice-Hall PTR, 1988.
62. Koelbel, Charles H., David B. Loveman, Robert S. Schreiber, Guy L. Steele, Jr., and Mary E. Zosel. *The High Performance Fortran Handbook.* Cambridge, MA: The MIT Press, 1994.
63. Lai, T. H., and S. Sahni. "Anomalies in parallel branch-and-bound algorithms." *Communications of the ACM* 27(6):594–602, June 1984.
64. Landau, David P., and Kurt Binder. *A Guide to Monte Carlo Simulations in Statistical Physics.* Cambridge: Cambridge University Press, 2000.
65. Landau, Rubin H., and Manuel J. Paez. *Computational Physics: Problem Solving with Computers.* New York: John Wiley & Sons, 1997.
66. Lawrie, Duncan H. "Access and alignment of data in an array processor." *IEEE Transactions on Computers* C-24(12):1145–1155, December 1975.
67. Lea, W. A. "Speech recognition: Past, present, and future." In *Trends in Speech Recognition.* Englewood Cliffs, NJ: Prentice-Hall, 1980.
68. L'Ecuyer, Pierre, and Richard Simard. "Beware of the linear congruential generators with multipliers of the form $a = \pm 2^q \pm 2^r$." *ACM Transactions on Mathematical Software* 25(3):367–374, September 1999.
69. Lehmer, Derrick Henry. "Mathematical methods in large scale computing units." In *Proceedings of the Second Symposium on Large-Scale Digital Calculating Machinery,* pages 141–146. Cambridge, MA: Harvard University Press, 1951.
70. Leiserson, Charles E. *Area-Efficient VLSI Computation.* Cambridge, MA: The MIT Press, 1983.
71. Lester, Bruce P. *The Art of Parallel Programming.* Englewood Cliffs, NJ: Prentice-Hall, 1993.
72. Leva, Joseph L. "A fast normal random number generator." *ACM Transactions on Mathematical Software* 18(4):449–453, December 1992.

73. Levin, E. "Grand challenges to computational science." *Communications of the ACM* 32(12):1456–1457, December 1989.

74. Li, X., P. Lu, J. Schaeffer, J. Shillington, P. S. Wong, and H. Shi. "On the versatility of parallel sorting by regular sampling." *Parallel Computing* 19:1079–1193, 1993.

75. Luhn, H. P. "The automatic creation of literature abstracts." *IBM Journal of Research and Development* 2(2):159–165, 317, April 1958.

76. Luo, Xuedong. "A practical sieve algorithm for finding prime numbers." *Communications of the ACM* 32(3):344–346, March 1989.

77. Makino, Jun. "Lagged-Fibonacci random number generators on parallel computers." *Parallel Computing* 20(9):1357–1367, 1994.

78. Manno, István. *Introduction to the Monte-Carlo Method.* Akadémiai Kiadoó, Budapest, Hungary, 1999.

79. Marsaglia, George. "Random numbers fall mainly in the planes." *Proceedings of the National Academy of Sciences of the United States of America* 62:25–28, 1968.

80. Marsaglia, George, and Wai Wan Tsang. "The Monty Python method for generating random variables." *ACM Transactions on Mathematical Software* 24(3):341–350, September 1998.

81. Marsland, T. A., and M. Campbell. "Parallel search of strongly ordered game trees." *Computing Surveys* 14(4):533–551, December 1982.

82. Mascagni, M., S. A. Cuccaro, D. V. Pryor, and M. L. Robinson. "A fast, high-quality, and reproducible lagged-Fibonacci pseudorandom number generator." *Computational Physics* 19:211–219, 1995.

83. Mascagni, M., and A. Srinivasan. "Algorithm 806. SPRNG: A scalable library for pseudorandom number generation." *ACM Transactions on Mathematical Software* 26(4):618–619, December 2000.

84. Mascagni, Michael. "Some methods of parallel pseudorandom number generation." In *Algorithms for Parallel Processing*, New York: Springer-Verlag, 1999.

85. McGraw, James R., and Timothy S. Axelrod. "Exploiting multiprocessors: Issues and options." In Robert G. Babb II, (ed.), *Programming Parallel Processors*, pages 7–25. Reading, MA: Addison-Wesley, 1988.

86. Mehrotra, Piyush, Joel Saltz, and Robert Voigt, editors. *Unstructured Scientific Computation on Scalable Multiprocessors.* Cambridge, MA: The MIT Press, 1992.

87. Moore, Gordon. "Cramming more components onto integrated circuits." *Electronics Magazine* 38(8):114–117, April 1965.

88. Nagendra, Bhavana, and Lars Rzymianowicz. "High speed networks." In Rajkumar Buyya, (ed.), *High Performance Cluster Computing: Architectures and Systems.* Vol. 1, pages 204–245. Upper Saddle River, NJ: Prentice-Hall PTR, 1999.

89. Pacheco, Peter S. *Parallel Programming with MPI.* San Francisco: Morgan Kaufmann, 1997.

90. Patterson, David A., and John L. Hennessy. *Computer Architecture: A Quantitative Approach.* 2d ed. San Francisco: Morgan Kaufmann, 1996.

91. Percus, Ora E., and Malvin H. Kalos. "Random number generators for MIMD parallel processors." *Journal of Parallel and Distributed Computing* 6:477–497, 1989.

92. Plybon, Benjamin F. *An Introduction to Applied Numerical Analysis.* Boston, MA: PWS-Kent Publishing Company, 1992.

93. Pountain, Dick, and David May. *A Tutorial Introduction to Occam Programming.* Oxford: BSP Professional Books, 1987.

94. Quinn, Michael J. "Parallel sorting algorithms for tightly coupled multiprocessors." *Parallel Computing* 6:349–357, 1988.

95. Quinn, Michael J. *Parallel Computing: Theory and Practice.* 2d ed. New York: McGraw-Hill, 1994.

96. Quinn, Michael J., and Narsingh Deo. "An upper bound for the speedup of parallel best-bound branch-and-bound algorithms." *BIT* 26(1):35–43, March 1986.

97. Reingold, E. M., J. Nievergelt, and N. Deo. *Combinatorial Algorithms: Theory and Practice.* Englewood Cliffs, NJ: Prentice-Hall, 1977.

98. Rumelhart, David E., Bernard Widrow, and Michael A. Lehr. "The basic ideas in neural networks." *Communications of the ACM* 37(3):87–92, March 1994.

99. Sabot, Gary W. (ed.). *High Performance Computing: Problem Solving with Parallel and Vector Architectures*. Reading, MA: Addison-Wesley, 1995.

100. Schaeffer, J. "Distributed game-tree searching." *Journal of Parallel and Distributed Computing* 6:90–114, 1989.

101. Shi, H., and J. Schaeffer. "Parallel sorting by regular sampling." *Journal of Parallel and Distributed Computing* 14:361–372, 1992.

102. Skillicorn, David B., and Domenico Talia. "Models and languages for parallel computation." *ACM Computing Surveys* 30(2):123–169, June 1998.

103. Slagle, J. R., and J. K. Dixon. "Experiments with some programs that search game trees." *Journal of the ACM* 16(2):189–207, April 1969.

104. Smith, G. D. *Numerical Solution of Partial Differential Equations: Finite Difference Methods*. Oxford: Oxford University Press, 1985.

105. Sterling, Thomas, (ed.). *Beowulf Cluster Computing with Linux*. Cambridge, MA: The MIT Press, 2002.

106. Tentner, A. M., R. N. Blomquist, T. R. Canfield, P. L. Garner, E. M. Gelbard, K. C. Gross, M. Minkoff, and R. A. Valentin. "Advances in parallel computing for reactor analysis and safety." *Communications of the ACM* 37(4):54–64, 1994.

107. Valiant, Leslie G. "A bridging model for parallel computation." *Communications of the ACM* 33(8), August 1990.

108. Wagar, Bruce. "Hyperquicksort: A fast sorting algorithm for hypercubes." In *Hypercube Multiprocessors 1987*, pages 292–299. Philadelphia, PA: SIAM, 1987.

109. Wah, B. W., G. Li, and C.-F. Yu. "Multiprocessing of combinatorial search problems." *Computer* 18(6):93–108, June 1985.

110. Wallace, C. S. "Fast pseudorandom generators for normal and exponential variables." *ACM Transactions on Mathematical Software* 22(1):119–127, March 1990.

111. Warshall, S. "A theorem on boolean matrices." *Journal of the ACM* 9(1):11–12, January 1962.

112. Weaver, H. J. *Applications of Discrete and Continuous Fourier Analysis*. New York: John Wiley & Sons, 1983.

113. Wheat, M., and D. J. Evans. "An efficient parallel sorting algorithm for shared memory multiprocessors." *Parallel Computing* 18:91–102, 1992.

114. Widrow, Bernard, David E. Rumelhart, and Michael A. Lehr. "Neural networks: Applications in industry, business, and science." *Communications of the ACM* 37(3):93–105, March 1994.

115. Wilkinson, Barry, and Michael Allen. *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers*. Upper Saddle River, NJ: Prentice-Hall, 1999.

116. Wilson, Gregory V. *Practical Parallel Programming*. Cambridge, MA: The MIT Press, 1995.

117. Wolfe, Michael. *High Performance Compilers for Parallel Computing*. Redwood City, CA: Addison-Wesley, 1996.

118. Wu, Pei-Chi. "Multiplicative, congruential random-number generators with multiplier $\pm 2^{k1} \pm 2^{k2}$ and modulus $2^p - 1$." *ACM Transactions on Mathematical Software* 23(2):255–265, June 1997.

AUTHOR INDEX

SUBJECT INDEX