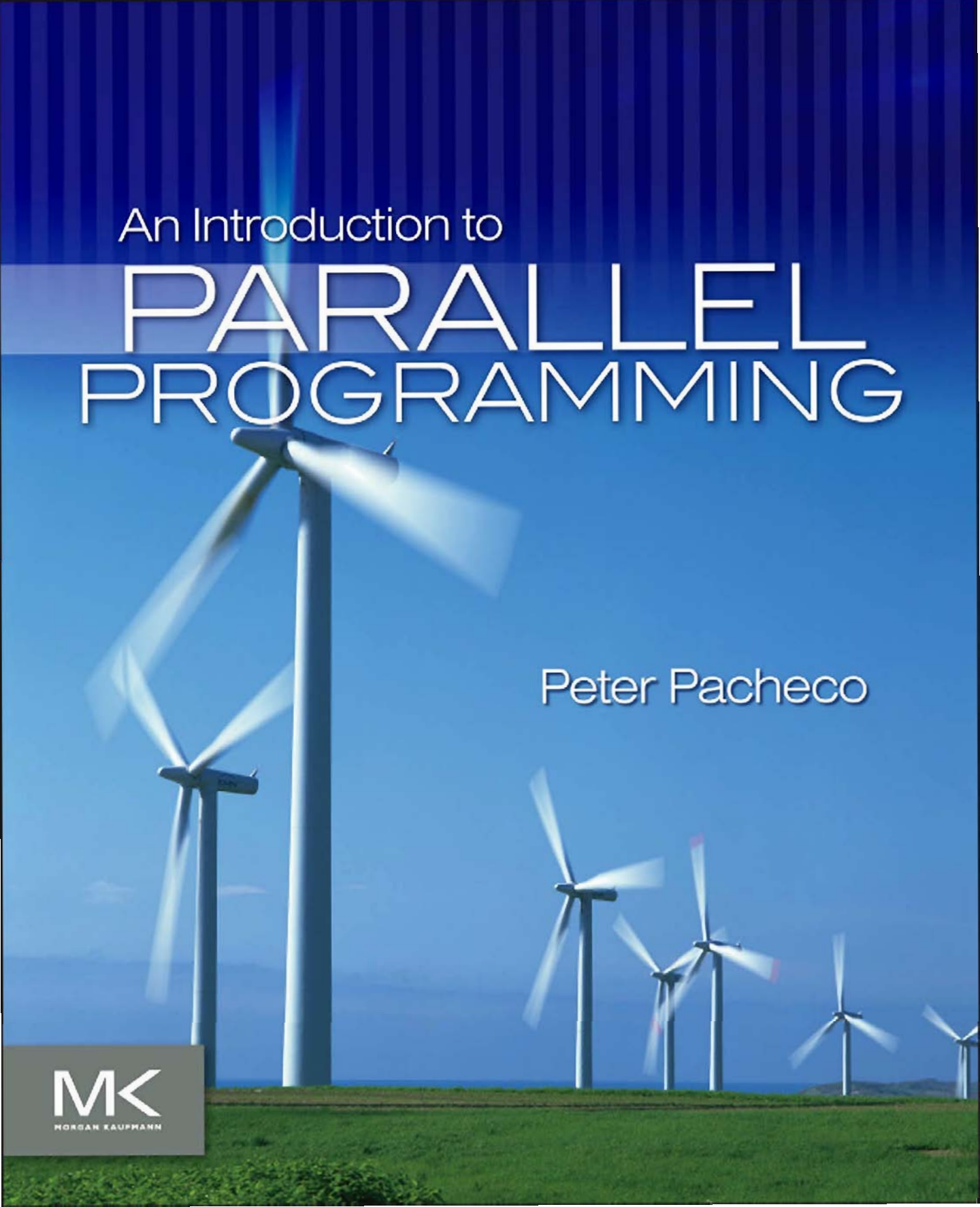
An Introduction to PARALLEL PROGRAMMING



Peter Pacheco

MK
MORGAN KAUFMANN

In Praise of *An Introduction to Parallel Programming*

With the coming of multicore processors and the cloud, parallel computing is most certainly not a niche area off in a corner of the computing world. Parallelism has become central to the efficient use of resources, and this new textbook by Peter Pacheco will go a long way toward introducing students early in their academic careers to both the art and practice of parallel computing.

Duncan Buell
Department of Computer Science and Engineering
University of South Carolina

An Introduction to Parallel Programming *illustrates fundamental programming principles in the increasingly important area of shared-memory programming using Pthreads and OpenMP and distributed-memory programming using MPI. More important, it emphasizes good programming practices by indicating potential performance pitfalls. These topics are presented in the context of a variety of disciplines, including computer science, physics, and mathematics. The chapters include numerous programming exercises that range from easy to very challenging. This is an ideal book for students or professionals looking to learn parallel programming skills or to refresh their knowledge.*

Leigh Little
Department of Computational Science
The College at Brockport, The State University of New York

An Introduction to Parallel Programming *is a well-written, comprehensive book on the field of parallel computing. Students and practitioners alike will appreciate the relevant, up-to-date information. Peter Pacheco's very accessible writing style, combined with numerous interesting examples, keeps the reader's attention. In a field that races forward at a dizzying pace, this book hangs on for the wild ride covering the ins and outs of parallel hardware and software.*

Kathy J. Liszka
Department of Computer Science
University of Akron

Parallel computing is the future and this book really helps introduce this complicated subject with practical and useful examples.

Andrew N. Sloss, FBCS
Consultant Engineer, ARM
Author of *ARM System Developer's Guide*

Recommended Reading List

For students interested in furthering their understanding of parallel programming, the content in the following books supplements this textbook:

Programming Massively Parallel Processors
A Hands-on Approach
By David B. Kirk and Wen-mei W. Hwu
ISBN: 9780123814722

The Art of Multiprocessor Programming
By Maurice Herlihy and Nir Shavit
ISBN: 9780123705914

Parallel Programming with MPI
By Peter Pacheco
ISBN: 9781558603394

The Sourcebook of Parallel Computing
Edited by Jack Dongarra et al.
ISBN: 9781558608719

Parallel Computer Architecture
A Hardware/Software Approach
By David Culler, J. P. Singh and Anoop Gupta
ISBN: 9781558603431

Engineering a Compiler, Second Edition
By Keith D. Cooper and Linda Torczon
ISBN: 9780120884780

mkp.com

An Introduction to Parallel Programming

This page intentionally left blank

An Introduction to Parallel Programming

Peter S. Pacheco
University of San Francisco

AMSTERDAM • BOSTON • HEIDELBERG • LONDON
NEW YORK • OXFORD • PARIS • SAN DIEGO
SAN FRANCISCO • SINGAPORE • SYDNEY • TOKYO

Morgan Kaufmann Publishers is an imprint of Elsevier

Acquiring Editor: Todd Green
Development Editor: Nate McFadden
Project Manager: Marilyn E. Rash
Designer: Joanne Blank

Morgan Kaufmann Publishers is an imprint of Elsevier.
30 Corporate Drive, Suite 400
Burlington, MA 01803, USA

Notices

Knowledge and best practice in this field are constantly changing. As new research and experience broaden our understanding, changes in research methods, professional practices, or medical treatment may become necessary.

Practitioners and researchers must always rely on their own experience and knowledge in evaluating and using any information, methods, compounds, or experiments described herein. In using such information or methods they should be mindful of their own safety and the safety of others, including parties for whom they have a professional responsibility.

To the fullest extent of the law, neither the Publisher nor the authors, contributors, or editors, assume any liability for any injury and/or damage to persons or property as a matter of products liability, negligence or otherwise, or from any use or operation of any methods, products, instructions, or ideas contained in the material herein.

Library of Congress Cataloging-in-Publication Data
Pacheco, Peter S.
An introduction to parallel programming / Peter S. Pacheco.
p. cm.
ISBN 978-0-12-374260-5 (hardback)
1. Parallel programming (Computer science) I. Title.
QA76.642.P29 2011
005.2′75–dc22

2010039584

British Library Cataloguing-in-Publication Data
A catalogue record for this book is available from the British Library.

For information on all Morgan Kaufmann publications,
visit our web site at *www.mkp.com* or *www.elsevierdirect.com*

Printed in the United States
11 12 13 14 15 10 9 8 7 6 5 4 3 2 1

To the memory of Josephine F. Pacheco

This page intentionally left blank

Contents

Preface xv
Acknowledgments xviii
About the Author xix

CHAPTER 1 Why Parallel Computing? 1
1.1 Why We Need Ever-Increasing Performance 2
1.2 Why We're Building Parallel Systems 3
1.3 Why We Need to Write Parallel Programs 3
1.4 How Do We Write Parallel Programs? 6
1.5 What We'll Be Doing 8
1.6 Concurrent, Parallel, Distributed 9
1.7 The Rest of the Book 10
1.8 A Word of Warning 10
1.9 Typographical Conventions 11
1.10 Summary 12
1.11 Exercises 12

CHAPTER 2 Parallel Hardware and Parallel Software 15
2.1 Some Background 15
2.1.1 The von Neumann architecture 15
2.1.2 Processes, multitasking, and threads 17
2.2 Modifications to the von Neumann Model 18
2.2.1 The basics of caching 19
2.2.2 Cache mappings 20
2.2.3 Caches and programs: an example 22
2.2.4 Virtual memory 23
2.2.5 Instruction-level parallelism 25
2.2.6 Hardware multithreading 28
2.3 Parallel Hardware 29
2.3.1 SIMD systems 29
2.3.2 MIMD systems 32
2.3.3 Interconnection networks 35
2.3.4 Cache coherence 43
2.3.5 Shared-memory versus distributed-memory 46
2.4 Parallel Software 47
2.4.1 Caveats 47
2.4.2 Coordinating the processes/threads 48
2.4.3 Shared-memory 49

2.4.4 Distributed-memory 53
2.4.5 Programming hybrid systems 56
2.5 Input and Output 56
2.6 Performance 58
2.6.1 Speedup and efficiency 58
2.6.2 Amdahl's law 61
2.6.3 Scalability 62
2.6.4 Taking timings 63
2.7 Parallel Program Design 65
2.7.1 An example 66
2.8 Writing and Running Parallel Programs 70
2.9 Assumptions 70
2.10 Summary 71
2.10.1 Serial systems 71
2.10.2 Parallel hardware 73
2.10.3 Parallel software 74
2.10.4 Input and output 75
2.10.5 Performance 75
2.10.6 Parallel program design 76
2.10.7 Assumptions 76
2.11 Exercises 77

CHAPTER 3 Distributed-Memory Programming with MPI 83
3.1 Getting Started 84
3.1.1 Compilation and execution 84
3.1.2 MPI programs 86
3.1.3 `MPI_Init` and `MPI_Finalize` 86
3.1.4 Communicators, `MPI_Comm_size` and `MPI_Comm_rank` 87
3.1.5 SPMD programs 88
3.1.6 Communication 88
3.1.7 `MPI_Send` 88
3.1.8 `MPI_Recv` 90
3.1.9 Message matching 91
3.1.10 The `status_p` argument 92
3.1.11 Semantics of `MPI_Send` and `MPI_Recv` 93
3.1.12 Some potential pitfalls 94
3.2 The Trapezoidal Rule in MPI 94
3.2.1 The trapezoidal rule 94
3.2.2 Parallelizing the trapezoidal rule 96

3.3 Dealing with I/O 97
3.3.1 Output 97
3.3.2 Input 100
3.4 Collective Communication 101
3.4.1 Tree-structured communication 102
3.4.2 `MPI_Reduce` 103
3.4.3 Collective vs. point-to-point communications 105
3.4.4 `MPI_Allreduce` 106
3.4.5 Broadcast 106
3.4.6 Data distributions 109
3.4.7 Scatter 110
3.4.8 Gather 112
3.4.9 Allgather 113
3.5 MPI Derived Datatypes 116
3.6 Performance Evaluation of MPI Programs 119
3.6.1 Taking timings 119
3.6.2 Results 122
3.6.3 Speedup and efficiency 125
3.6.4 Scalability 126
3.7 A Parallel Sorting Algorithm 127
3.7.1 Some simple serial sorting algorithms 127
3.7.2 Parallel odd-even transposition sort 129
3.7.3 Safety in MPI programs 132
3.7.4 Final details of parallel odd-even sort 134
3.8 Summary 136
3.9 Exercises 140
3.10 Programming Assignments 147

CHAPTER 4 Shared-Memory Programming with Pthreads 151
4.1 Processes, Threads, and Pthreads 151
4.2 Hello, World 153
4.2.1 Execution 153
4.2.2 Preliminaries 155
4.2.3 Starting the threads 156
4.2.4 Running the threads 157
4.2.5 Stopping the threads 158
4.2.6 Error checking 158
4.2.7 Other approaches to thread startup 159
4.3 Matrix-Vector Multiplication 159
4.4 Critical Sections 162

4.5 Busy-Waiting 165
4.6 Mutexes 168
4.7 Producer-Consumer Synchronization and Semaphores..... 171
4.8 Barriers and Condition Variables..... 176
4.8.1 Busy-waiting and a mutex 177
4.8.2 Semaphores 177
4.8.3 Condition variables 179
4.8.4 Pthreads barriers 181
4.9 Read-Write Locks 181
4.9.1 Linked list functions..... 181
4.9.2 A multi-threaded linked list..... 183
4.9.3 Pthreads read-write locks 187
4.9.4 Performance of the various implementations 188
4.9.5 Implementing read-write locks 190
4.10 Caches, Cache Coherence, and False Sharing 190
4.11 Thread-Safety..... 195
4.11.1 Incorrect programs can produce correct output..... 198
4.12 Summary 198
4.13 Exercises 200
4.14 Programming Assignments 206

CHAPTER 5 Shared-Memory Programming with OpenMP 209
5.1 Getting Started..... 210
5.1.1 Compiling and running OpenMP programs..... 211
5.1.2 The program 212
5.1.3 Error checking 215
5.2 The Trapezoidal Rule..... 216
5.2.1 A first OpenMP version 216
5.3 Scope of Variables 220
5.4 The Reduction Clause 221
5.5 The `parallel for` Directive 224
5.5.1 Caveats 225
5.5.2 Data dependences..... 227
5.5.3 Finding loop-carried dependences..... 228
5.5.4 Estimating π 229
5.5.5 More on scope 231
5.6 More About Loops in OpenMP: Sorting 232
5.6.1 Bubble sort 232
5.6.2 Odd-even transposition sort..... 233
5.7 Scheduling Loops 236
5.7.1 The `schedule` clause..... 237
5.7.2 The `static` schedule type..... 238

5.7.3 The `dynamic` and `guided` schedule types 239
5.7.4 The `runtime` schedule type 239
5.7.5 Which schedule? 241
5.8 Producers and Consumers 241
5.8.1 Queues 241
5.8.2 Message-passing 242
5.8.3 Sending messages 243
5.8.4 Receiving messages 243
5.8.5 Termination detection 244
5.8.6 Startup 244
5.8.7 The `atomic` directive 245
5.8.8 Critical sections and locks 246
5.8.9 Using locks in the message-passing program 248
5.8.10 `critical` directives, `atomic` directives, or locks? 249
5.8.11 Some caveats 249
5.9 Caches, Cache Coherence, and False Sharing 251
5.10 Thread-Safety 256
5.10.1 Incorrect programs can produce correct output 258
5.11 Summary 259
5.12 Exercises 263
5.13 Programming Assignments 267

CHAPTER 6 Parallel Program Development 271
6.1 Two *n*-Body Solvers 271
6.1.1 The problem 271
6.1.2 Two serial programs 273
6.1.3 Parallelizing the *n*-body solvers 277
6.1.4 A word about I/O 280
6.1.5 Parallelizing the basic solver using OpenMP 281
6.1.6 Parallelizing the reduced solver using OpenMP 284
6.1.7 Evaluating the OpenMP codes 288
6.1.8 Parallelizing the solvers using pthreads 289
6.1.9 Parallelizing the basic solver using MPI 290
6.1.10 Parallelizing the reduced solver using MPI 292
6.1.11 Performance of the MPI solvers 297
6.2 Tree Search 299
6.2.1 Recursive depth-first search 302
6.2.2 Nonrecursive depth-first search 303
6.2.3 Data structures for the serial implementations 305
6.2.4 Performance of the serial implementations 306
6.2.5 Parallelizing tree search 306

6.2.6 A static parallelization of tree search using pthreads 309
6.2.7 A dynamic parallelization of tree search using pthreads 310
6.2.8 Evaluating the pthreads tree-search programs 315
6.2.9 Parallelizing the tree-search programs using OpenMP 316
6.2.10 Performance of the OpenMP implementations 318
6.2.11 Implementation of tree search using MPI and static partitioning 319
6.2.12 Implementation of tree search using MPI and dynamic partitioning 327
6.3 A Word of Caution 335
6.4 Which API? 335
6.5 Summary 336
6.5.1 Pthreads and OpenMP 337
6.5.2 MPI 338
6.6 Exercises 341
6.7 Programming Assignments 350

CHAPTER 7 Where to Go from Here 353

References 357

Index 361

Preface

Parallel hardware has been ubiquitous for some time now. It's difficult to find a laptop, desktop, or server that doesn't use a multicore processor. Beowulf clusters are nearly as common today as high-powered workstations were during the 1990s, and cloud computing could make distributed-memory systems as accessible as desktops. In spite of this, most computer science majors graduate with little or no experience in parallel programming. Many colleges and universities offer upper-division elective courses in parallel computing, but since most computer science majors have to take numerous required courses, many graduate without ever writing a multithreaded or multiprocess program.

It seems clear that this state of affairs needs to change. Although many programs can obtain satisfactory performance on a single core, computer scientists should be made aware of the potentially vast performance improvements that can be obtained with parallelism, and they should be able to exploit this potential when the need arises.

An Introduction to Parallel Programming was written to partially address this problem. It provides an introduction to writing parallel programs using MPI, Pthreads, and OpenMP—three of the most widely used application programming interfaces (APIs) for parallel programming. The intended audience is students and professionals who need to write parallel programs. The prerequisites are minimal: a college-level course in mathematics and the ability to write serial programs in C. They are minimal because we believe that students should be able to start programming parallel systems *as early as possible.*

At the University of San Francisco, computer science students can fulfill a requirement for the major by taking the course, on which this text is based, immediately after taking the "Introduction to Computer Science I" course that most majors take in the first semester of their freshman year. We've been offering this course in parallel computing for six years now, and it has been our experience that there really is no reason for students to defer writing parallel programs until their junior or senior year. To the contrary, the course is popular, and students have found that using concurrency in other courses is much easier after having taken the Introduction course.

If second-semester freshmen can learn to write parallel programs by taking a class, then motivated computing professionals should be able to learn to write parallel programs through self-study. We hope this book will prove to be a useful resource for them.

About This Book

As we noted earlier, the main purpose of the book is to teach parallel programming in MPI, Pthreads, and OpenMP to an audience with a limited background in computer science and no previous experience with parallelism. We also wanted to make it as

flexible as possible so that readers who have no interest in learning one or two of the APIs can still read the remaining material with little effort. Thus, the chapters on the three APIs are largely independent of each other: they can be read in any order, and one or two of these chapters can be bypass. This independence has a cost: It was necessary to repeat some of the material in these chapters. Of course, repeated material can be simply scanned or skipped.

Readers with no prior experience with parallel computing should read Chapter 1 first. It attempts to provide a relatively nontechnical explanation of why parallel systems have come to dominate the computer landscape. The chapter also provides a short introduction to parallel systems and parallel programming.

Chapter 2 provides some technical background in computer hardware and software. Much of the material on hardware can be scanned before proceeding to the API chapters. Chapters 3, 4, and 5 are the introductions to programming with MPI, Pthreads, and OpenMP, respectively.

In Chapter 6 we develop two longer programs: a parallel *n*-body solver and a parallel tree search. Both programs are developed using all three APIs. Chapter 7 provides a brief list of pointers to additional information on various aspects of parallel computing.

We use the C programming language for developing our programs because all three APIs have C-language interfaces, and, since C is such a small language, it is a relatively easy language to learn—especially for C++ and Java programmers, since they are already familiar with C's control structures.

Classroom Use

This text grew out of a lower-division undergraduate course at the University of San Francisco. The course fulfills a requirement for the computer science major, and it also fulfills a prerequisite for the undergraduate operating systems course. The only prerequisites for the course are either a grade of "B" or better in a one-semester introduction to computer science or a "C" or better in a two-semester introduction to computer science. The course begins with a four-week introduction to C programming. Since most students have already written Java programs, the bulk of what is covered is devoted to the use pointers in C.[1] The remainder of the course provides introductions to programming in MPI, Pthreads, and OpenMP.

We cover most of the material in Chapters 1, 3, 4, and 5, and parts of the material in Chapters 2 and 6. The background in Chapter 2 is introduced as the need arises. For example, before discussing cache coherence issues in OpenMP (Chapter 5), we cover the material on caches in Chapter 2.

The coursework consists of weekly homework assignments, five programming assignments, a couple of midterms, and a final exam. The homework usually involves

[1] Interestingly, a number of students have said that they found the use of C pointers more difficult than MPI programming.

writing a very short program or making a small modification to an existing program. Their purpose is to insure that students stay current with the course work and to give them hands-on experience with the ideas introduced in class. It seems likely that the existence of the assignments has been one of the principle reasons for the course's success. Most of the exercises in the text are suitable for these brief assignments.

The programming assignments are larger than the programs written for homework, but we typically give students a good deal of guidance: We'll frequently include pseudocode in the assignment and discuss some of the more difficult aspects in class. This extra guidance is often crucial: It's not difficult to give programming assignments that will take far too long for the students to complete. The results of the midterms and finals, and the enthusiastic reports of the professor who teaches operating systems, suggest that the course is actually very successful in teaching students how to write parallel programs.

For more advanced courses in parallel computing, the text and its online support materials can serve as a supplement so that much of the information on the syntax and semantics of the three APIs can be assigned as outside reading. The text can also be used as a supplement for project-based courses and courses outside of computer science that make use of parallel computation.

Support Materials

The book's website is located at `http://www.mkp.com/pacheco`. It will include errata and links to sites with related materials. Faculty will be able to download complete lecture notes, figures from the text, and solutions to the exercises and the programming assignments. All users will be able to download the longer programs discussed in the text.

We would greatly appreciate readers letting us know of any errors they find. Please send an email to `peter@usfca.edu` if you do find a mistake.

Acknowledgments

In the course of working on this book, I've received considerable help from many individuals. Among them I'd like to thank the reviewers who read and commented on the initial proposal: Fikret Ercal (Missouri University of Science and Technology), Dan Harvey (Southern Oregon University), Joel Hollingsworth (Elon University), Jens Mache (Lewis and Clark College), Don McLaughlin (West Virginia University), Manish Parashar (Rutgers University), Charlie Peck (Earlham College), Stephen C. Renk (North Central College), Rolfe Josef Sassenfeld (The University of Texas at El Paso), Joseph Sloan (Wofford College), Michela Taufer (University of Delaware), Pearl Wang (George Mason University), Bob Weems (University of Texas at Arlington), and Cheng-Zhong Xu (Wayne State University).

I'm also deeply grateful to the following individuals for their reviews of various chapters of the book: Duncan Buell (University of South Carolina), Matthias Gobbert (University of Maryland, Baltimore County), Krishna Kavi (University of North Texas), Hong Lin (University of Houston–Downtown), Kathy Liszka (University of Akron), Leigh Little (The State University of New York), Xinlian Liu (Hood College), Henry Tufo (University of Colorado at Boulder), Andrew Sloss (Consultant Engineer, ARM), and Gengbin Zheng (University of Illinois). Their comments and suggestions have made the book immeasurably better. Of course, I'm solely responsible for remaining errors and omissions.

Kathy Liszka is also preparing slides that can be used by faculty who adopt the text, and a former student, Jinyoung Choi, is working on preparing a solutions manual. Thanks to both of them.

The staff of Morgan Kaufmann has been very helpful throughout this project. I'm especially grateful to the developmental editor, Nate McFadden. He gave me much valuable advice, and he did a terrific job arranging for the reviews. He's also been tremendously patient with all the problems I've encountered over the past few years. Thanks also to Marilyn Rash and Megan Guiney, who have been very prompt and efficient during the production process.

My colleagues in the computer science and mathematics departments at USF have been extremely helpful during my work on the book. I'd like to single out Professor Gregory Benson for particular thanks: his understanding of parallel computing—especially Pthreads and semaphores—has been an invaluable resource for me. I'm also very grateful to our system administrators, Alexey Fedosov and Colin Bean. They've patiently and efficiently dealt with all of the "emergencies" that cropped up while I was working on programs for the book.

I would never have been able to finish this book without the encouragement and moral support of my friends Holly Cohn, John Dean, and Robert Miller. They helped me through some very difficult times, and I'll be eternally grateful to them.

My biggest debt is to my students. They showed me what was too easy, and what was far too difficult. In short, they taught me how to teach parallel computing. My deepest thanks to all of them.

About the Author

Peter Pacheco received a PhD in mathematics from Florida State University. After completing graduate school, he became one of the first professors in UCLA's "Program in Computing," which teaches basic computer science to students at the College of Letters and Sciences there. Since leaving UCLA, he has been on the faculty of the University of San Francisco. At USF Peter has served as chair of the computer science department and is currently chair of the mathematics department.

His research is in parallel scientific computing. He has worked on the development of parallel software for circuit simulation, speech recognition, and the simulation of large networks of biologically accurate neurons. Peter has been teaching parallel computing at both the undergraduate and graduate levels for nearly twenty years. He is the author of *Parallel Programming with MPI*, published by Morgan Kaufmann Publishers.

This page intentionally left blank

CHAPTER

1 Why Parallel Computing?

From 1986 to 2002 the performance of microprocessors increased, on average, 50% per year [27]. This unprecedented increase meant that users and software developers could often simply wait for the next generation of microprocessors in order to obtain increased performance from an application program. Since 2002, however, single-processor performance improvement has slowed to about 20% per year. This difference is dramatic: at 50% per year, performance will increase by almost a factor of 60 in 10 years, while at 20%, it will only increase by about a factor of 6.

Furthermore, this difference in performance increase has been associated with a dramatic change in processor design. By 2005, most of the major manufacturers of microprocessors had decided that the road to rapidly increasing performance lay in the direction of parallelism. Rather than trying to continue to develop ever-faster monolithic processors, manufacturers started putting *multiple* complete processors on a single integrated circuit.

This change has a very important consequence for software developers: simply adding more processors will not magically improve the performance of the vast majority of **serial** programs, that is, programs that were written to run on a single processor. Such programs are unaware of the existence of multiple processors, and the performance of such a program on a system with multiple processors will be effectively the same as its performance on a single processor of the multiprocessor system.

All of this raises a number of questions:

1. Why do we care? Aren't single processor systems fast enough? After all, 20% per year is still a pretty significant performance improvement.
2. Why can't microprocessor manufacturers continue to develop much faster single processor systems? Why build **parallel systems**? Why build systems with multiple processors?
3. Why can't we write programs that will automatically convert serial programs into **parallel programs**, that is, programs that take advantage of the presence of multiple processors?

Let's take a brief look at each of these questions. Keep in mind, though, that some of the answers aren't carved in stone. For example, 20% per year may be more than adequate for many applications.

An Introduction to Parallel Programming. DOI: 10.1016/B978-0-12-374260-5.00001-4

1.1 WHY WE NEED EVER-INCREASING PERFORMANCE

The vast increases in computational power that we've been enjoying for decades now have been at the heart of many of the most dramatic advances in fields as diverse as science, the Internet, and entertainment. For example, decoding the human genome, ever more accurate medical imaging, astonishingly fast and accurate Web searches, and ever more realistic computer games would all have been impossible without these increases. Indeed, more recent increases in computational power would have been difficult, if not impossible, without earlier increases. But we can never rest on our laurels. As our computational power increases, the number of problems that we can seriously consider solving also increases. The following are a few examples:

- *Climate modeling*. In order to better understand climate change, we need far more accurate computer models, models that include interactions between the atmosphere, the oceans, solid land, and the ice caps at the poles. We also need to be able to make detailed studies of how various interventions might affect the global climate.
- *Protein folding*. It's believed that misfolded proteins may be involved in diseases such as Huntington's, Parkinson's, and Alzheimer's, but our ability to study configurations of complex molecules such as proteins is severely limited by our current computational power.
- *Drug discovery*. There are many ways in which increased computational power can be used in research into new medical treatments. For example, there are many drugs that are effective in treating a relatively small fraction of those suffering from some disease. It's possible that we can devise alternative treatments by careful analysis of the genomes of the individuals for whom the known treatment is ineffective. This, however, will involve extensive computational analysis of genomes.
- *Energy research*. Increased computational power will make it possible to program much more detailed models of technologies such as wind turbines, solar cells, and batteries. These programs may provide the information needed to construct far more efficient clean energy sources.
- *Data analysis*. We generate tremendous amounts of data. By some estimates, the quantity of data stored worldwide doubles every two years [28], but the vast majority of it is largely useless unless it's analyzed. As an example, knowing the sequence of nucleotides in human DNA is, by itself, of little use. Understanding how this sequence affects development and how it can cause disease requires extensive analysis. In addition to genomics, vast quantities of data are generated by particle colliders such as the Large Hadron Collider at CERN, medical imaging, astronomical research, and Web search engines—to name a few.

These and a host of other problems won't be solved without vast increases in computational power.

1.2 WHY WE'RE BUILDING PARALLEL SYSTEMS

Much of the tremendous increase in single processor performance has been driven by the ever-increasing density of transistors—the electronic switches—on integrated circuits. As the size of transistors decreases, their speed can be increased, and the overall speed of the integrated circuit can be increased. However, as the speed of transistors increases, their power consumption also increases. Most of this power is dissipated as heat, and when an integrated circuit gets too hot, it becomes unreliable. In the first decade of the twenty-first century, air-cooled integrated circuits are reaching the limits of their ability to dissipate heat [26].

Therefore, it is becoming impossible to continue to increase the speed of integrated circuits. However, the increase in transistor density *can* continue—at least for a while. Also, given the potential of computing to improve our existence, there is an almost moral imperative to continue to increase computational power. Finally, if the integrated circuit industry doesn't continue to bring out new and better products, it will effectively cease to exist.

How then, can we exploit the continuing increase in transistor density? The answer is *parallelism.* Rather than building ever-faster, more complex, monolithic processors, the industry has decided to put multiple, relatively simple, complete processors on a single chip. Such integrated circuits are called **multicore** processors, and **core** has become synonymous with central processing unit, or CPU. In this setting a conventional processor with one CPU is often called a **single-core** system.

1.3 WHY WE NEED TO WRITE PARALLEL PROGRAMS

Most programs that have been written for conventional, single-core systems cannot exploit the presence of multiple cores. We can run multiple instances of a program on a multicore system, but this is often of little help. For example, being able to run multiple instances of our favorite game program isn't really what we want—we want the program to run faster with more realistic graphics. In order to do this, we need to either rewrite our serial programs so that they're *parallel*, so that they can make use of multiple cores, or write translation programs, that is, programs that will automatically convert serial programs into parallel programs. The bad news is that researchers have had very limited success writing programs that convert serial programs in languages such as C and C++ into parallel programs.

This isn't terribly surprising. While we can write programs that recognize common constructs in serial programs, and automatically translate these constructs into efficient parallel constructs, the sequence of parallel constructs may be terribly inefficient. For example, we can view the multiplication of two $n \times n$ matrices as a sequence of dot products, but parallelizing a matrix multiplication as a sequence of parallel dot products is likely to be very slow on many systems.

An efficient parallel implementation of a serial program may not be obtained by finding efficient parallelizations of each of its steps. Rather, the best parallelization may be obtained by stepping back and devising an entirely new algorithm.

As an example, suppose that we need to compute n values and add them together. We know that this can be done with the following serial code:

```
sum = 0;
for (i = 0; i < n; i++) {
   x = Compute_next_value(. . .);
   sum += x;
}
```

Now suppose we also have p cores and p is much smaller than n. Then each core can form a partial sum of approximately n/p values:

```
my_sum = 0;
my_first_i = . . . ;
my_last_i = . . . ;
for (my_i = my_first_i; my_i < my_last_i; my_i++) {
   my_x = Compute_next_value(. . .);
   my_sum += my_x;
}
```

Here the prefix `my_` indicates that each core is using its own, private variables, and each core can execute this block of code independently of the other cores.

After each core completes execution of this code, its variable `my_sum` will store the sum of the values computed by its calls to `Compute_next_value`. For example, if there are eight cores, $n = 24$, and the 24 calls to `Compute_next_value` return the values

$$1,4,3,\ 9,2,8,\ 5,1,1,\ 6,2,7,\ 2,5,0,\ 4,1,8,\ 6,5,1,\ 2,3,9,$$

then the values stored in `my_sum` might be

Core	0	1	2	3	4	5	6	7
my_sum	8	19	7	15	7	13	12	14

Here we're assuming the cores are identified by nonnegative integers in the range $0, 1, \ldots, p-1$, where p is the number of cores.

When the cores are done computing their values of `my_sum`, they can form a global sum by sending their results to a designated "master" core, which can add their results:

```
if (I'm the master core) {
   sum = my_x;
   for each core other than myself {
      receive value from core;
      sum += value;
   }
```

```
} else {
   send my_x to the master;
}
```

In our example, if the master core is core 0, it would add the values $8 + 19 + 7 + 15 + 7 + 13 + 12 + 14 = 95$.

But you can probably see a better way to do this—especially if the number of cores is large. Instead of making the master core do all the work of computing the final sum, we can pair the cores so that while core 0 adds in the result of core 1, core 2 can add in the result of core 3, core 4 can add in the result of core 5 and so on. Then we can repeat the process with only the even-ranked cores: 0 adds in the result of 2, 4 adds in the result of 6, and so on. Now cores divisible by 4 repeat the process, and so on. See Figure 1.1. The circles contain the current value of each core's sum, and the lines with arrows indicate that one core is sending its sum to another core. The plus signs indicate that a core is receiving a sum from another core and adding the received sum into its own sum.

For both "global" sums, the master core (core 0) does more work than any other core, and the length of time it takes the program to complete the final sum should be the length of time it takes for the master to complete. However, with eight cores, the master will carry out seven receives and adds using the first method, while with the second method it will only carry out three. So the second method results in an improvement of more than a factor of two. The difference becomes much more

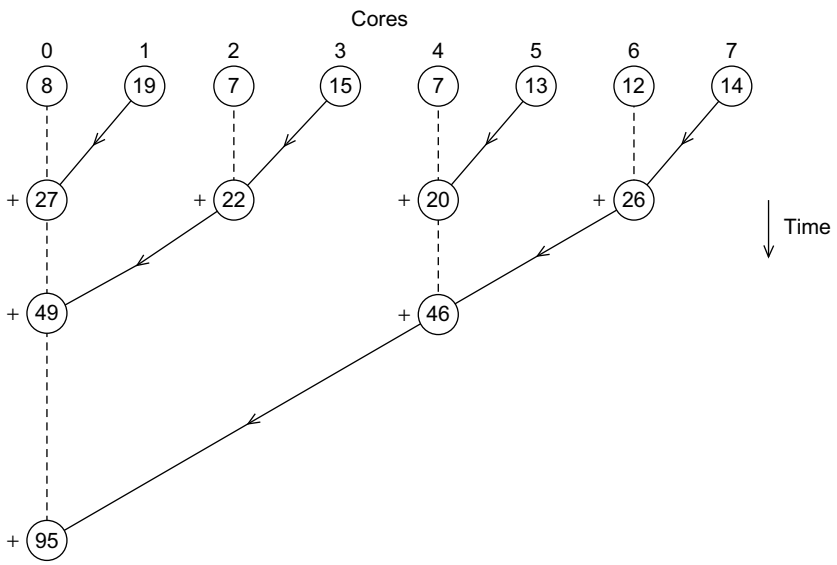


FIGURE 1.1

Multiple cores forming a global sum

dramatic with large numbers of cores. With 1000 cores, the first method will require 999 receives and adds, while the second will only require 10, an improvement of almost a factor of 100!

The first global sum is a fairly obvious generalization of the serial global sum: divide the work of adding among the cores, and after each core has computed its part of the sum, the master core simply repeats the basic serial addition—if there are p cores, then it needs to add p values. The second global sum, on the other hand, bears little relation to the original serial addition.

The point here is that it's unlikely that a translation program would "discover" the second global sum. Rather there would more likely be a predefined efficient global sum that the translation program would have access to. It could "recognize" the original serial loop and replace it with a precoded, efficient, parallel global sum.

We might expect that software could be written so that a large number of common serial constructs could be recognized and efficiently **parallelized**, that is, modified so that they can use multiple cores. However, as we apply this principle to ever more complex serial programs, it becomes more and more difficult to recognize the construct, and it becomes less and less likely that we'll have a precoded, efficient parallelization.

Thus, we cannot simply continue to write serial programs, we must write parallel programs, programs that exploit the power of multiple processors.

1.4 HOW DO WE WRITE PARALLEL PROGRAMS?

There are a number of possible answers to this question, but most of them depend on the basic idea of *partitioning* the work to be done among the cores. There are two widely used approaches: **task-parallelism** and **data-parallelism**. In task-parallelism, we partition the various tasks carried out in solving the problem among the cores. In data-parallelism, we partition the data used in solving the problem among the cores, and each core carries out more or less similar operations on its part of the data.

As an example, suppose that Prof P has to teach a section of "Survey of English Literature." Also suppose that Prof P has one hundred students in her section, so she's been assigned four teaching assistants (TAs): Mr A, Ms B, Mr C, and Ms D. At last the semester is over, and Prof P makes up a final exam that consists of five questions. In order to grade the exam, she and her TAs might consider the following two options: each of them can grade all one hundred responses to one of the questions; say P grades question 1, A grades question 2, and so on. Alternatively, they can divide the one hundred exams into five piles of twenty exams each, and each of them can grade all the papers in one of the piles; P grades the papers in the first pile, A grades the papers in the second pile, and so on.

In both approaches the "cores" are the professor and her TAs. The first approach might be considered an example of task-parallelism. There are five tasks to be carried out: grading the first question, grading the second question, and so on. Presumably, the graders will be looking for different information in question 1, which is about

Shakespeare, from the information in question 2, which is about Milton, and so on. So the professor and her TAs will be "executing different instructions."

On the other hand, the second approach might be considered an example of data-parallelism. The "data" are the students' papers, which are divided among the cores, and each core applies more or less the same grading instructions to each paper.

The first part of the global sum example in Section 1.3 would probably be considered an example of data-parallelism. The data are the values computed by `Compute_next_value`, and each core carries out roughly the same operations on its assigned elements: it computes the required values by calling `Compute_next_value` and adds them together. The second part of the first global sum example might be considered an example of task-parallelism. There are two tasks: receiving and adding the cores' partial sums, which is carried out by the master core, and giving the partial sum to the master core, which is carried out by the other cores.

When the cores can work independently, writing a parallel program is much the same as writing a serial program. Things get a good deal more complex when the cores need to coordinate their work. In the second global sum example, although the tree structure in the diagram is very easy to understand, writing the actual code is relatively complex. See Exercises 1.3 and 1.4. Unfortunately, it's much more common for the cores to need coordination.

In both global sum examples, the coordination involves **communication**: one or more cores send their current partial sums to another core. The global sum examples should also involve coordination through **load balancing**: even though we didn't give explicit formulas, it's clear that we want the cores all to be assigned roughly the same number of values to compute. If, for example, one core has to compute most of the values, then the other cores will finish much sooner than the heavily loaded core, and their computational power will be wasted.

A third type of coordination is **synchronization**. As an example, suppose that instead of computing the values to be added, the values are read from `stdin`. Say `x` is an array that is read in by the master core:

```
if (I'm the master core)
   for (my_i = 0; my_i < n; my_i++)
      scanf("%lf", &x[my_i]);
```

In most systems the cores are not automatically synchronized. Rather, each core works at its own pace. In this case, the problem is that we don't want the other cores to race ahead and start computing their partial sums before the master is done initializing `x` and making it available to the other cores. That is, the cores need to wait before starting execution of the code:

```
for (my_i = my_first_i; my_i < my_last_i; my_i++)
   my_sum += x[my_i];
```

We need to add in a point of synchronization between the initialization of `x` and the computation of the partial sums:

```
Synchronize_cores();
```

The idea here is that each core will wait in the function `Synchronize_cores` until all the cores have entered the function—in particular, until the master core has entered this function.

Currently, the most powerful parallel programs are written using *explicit* parallel constructs, that is, they are written using extensions to languages such as C and C++. These programs include explicit instructions for parallelism: core 0 executes task 0, core 1 executes task 1, . . . , all cores synchronize, . . . , and so on, so such programs are often extremely complex. Furthermore, the complexity of modern cores often makes it necessary to use considerable care in writing the code that will be executed by a single core.

There are other options for writing parallel programs—for example, higher level languages—but they tend to sacrifice performance in order to make program development somewhat easier.

1.5 WHAT WE'LL BE DOING

We'll be focusing on learning to write programs that are *explicitly* parallel. Our purpose is to learn the basics of programming parallel computers using the C language and three different extensions to C: the **Message-Passing Interface** or **MPI, POSIX threads** or **Pthreads,** and **OpenMP**. MPI and Pthreads are libraries of type definitions, functions, and macros that can be used in C programs. OpenMP consists of a library and some modifications to the C compiler.

You may well wonder why we're learning three different extensions to C instead of just one. The answer has to do with both the extensions and parallel systems. There are two main types of parallel systems that we'll be focusing on: **shared-memory** systems and **distributed-memory** systems. In a shared-memory system, the cores can share access to the computer's memory; in principle, each core can read and write each memory location. In a shared-memory system, we can coordinate the cores by having them examine and update shared-memory locations. In a distributed-memory system, on the other hand, each core has its own, private memory, and the cores must communicate explicitly by doing something like sending messages across a network. Figure 1.2 shows a schematic of the two types of systems. Pthreads and OpenMP were designed for programming shared-memory systems. They provide mechanisms for accessing shared-memory locations. MPI, on the other hand, was designed for programming distributed-memory systems. It provides mechanisms for sending messages.

But why two extensions for shared-memory? OpenMP is a relatively high-level extension to C. For example, it can "parallelize" our addition loop

```
sum = 0;
for (i = 0; i < n; i++) {
   x = Compute_next_value(. . .);
   sum += x;
}
```

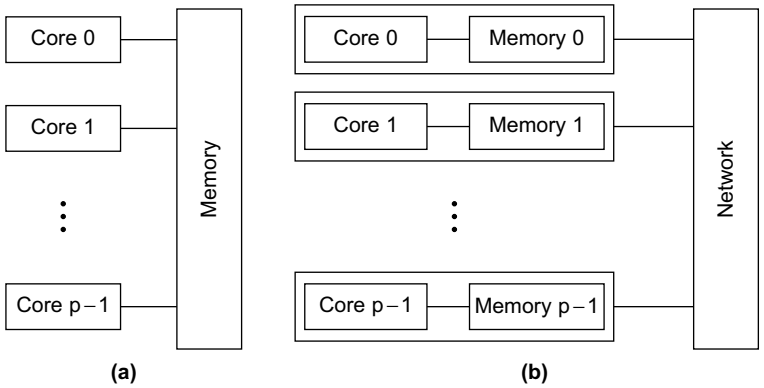


FIGURE 1.2

(a) A shared-memory system and (b) a distributed-memory system

with a single directive, while Pthreads requires that we do something similar to our example. On the other hand, Pthreads provides some coordination constructs that are unavailable in OpenMP. OpenMP allows us to parallelize many programs with relative ease, while Pthreads provides us with some constructs that make other programs easier to parallelize.

1.6 CONCURRENT, PARALLEL, DISTRIBUTED

If you look at some other books on parallel computing or you search the Web for information on parallel computing, you're likely to also run across the terms **concurrent computing** and **distributed computing**. Although there isn't complete agreement on the distinction between the terms parallel, distributed, and concurrent, many authors make the following distinctions:

- In concurrent computing, a program is one in which multiple tasks can be *in progress* at any instant [4].
- In parallel computing, a program is one in which multiple tasks *cooperate closely* to solve a problem.
- In distributed computing, a program may need to cooperate with other programs to solve a problem.

So parallel and distributed programs are concurrent, but a program such as a multitasking operating system is also concurrent, even when it is run on a machine with only one core, since multiple tasks can be *in progress* at any instant. There isn't a clear-cut distinction between parallel and distributed programs, but a parallel program usually runs multiple tasks simultaneously on cores that are physically close to each other and that either share the same memory or are connected by a very high-speed

network. On the other hand, distributed programs tend to be more "loosely coupled." The tasks may be executed by multiple computers that are separated by large distances, and the tasks themselves are often executed by programs that were created independently. As examples, our two concurrent addition programs would be considered parallel by most authors, while a Web search program would be considered distributed.

But beware, there isn't general agreement on these terms. For example, many authors consider shared-memory programs to be "parallel" and distributed-memory programs to be "distributed." As our title suggests, we'll be interested in parallel programs—programs in which closely coupled tasks cooperate to solve a problem.

1.7 THE REST OF THE BOOK

How can we use this book to help us write parallel programs?

First, when you're interested in high-performance, whether you're writing serial or parallel programs, you need to know a little bit about the systems you're working with—both hardware and software. In Chapter 2, we'll give an overview of parallel hardware and software. In order to understand this discussion, it will be necessary to review some information on serial hardware and software. Much of the material in Chapter 2 won't be needed when we're getting started, so you might want to skim some of this material, and refer back to it occasionally when you're reading later chapters.

The heart of the book is contained in Chapters 3 through 6. Chapters 3, 4, and 5 provide a very elementary introduction to programming parallel systems using C and MPI, Pthreads, and OpenMP, respectively. The only prerequisite for reading these chapters is a knowledge of C programming. We've tried to make these chapters independent of each other, and you should be able to read them in any order. However, in order to make them independent, we did find it necessary to repeat some material. So if you've read one of the three chapters, and you go on to read another, be prepared to skim over some of the material in the new chapter.

Chapter 6 puts together all we've learned in the preceding chapters, and develops two fairly large programs in both a shared- and a distributed-memory setting. However, it should be possible to read much of this even if you've only read one of Chapters 3, 4, or 5. The last chapter, Chapter 7, provides a few suggestions for further study on parallel programming.

1.8 A WORD OF WARNING

Before proceeding, a word of warning. It may be tempting to write parallel programs "by the seat of your pants," without taking the trouble to carefully design and incrementally develop your program. This will almost certainly be a mistake. Every

parallel program contains at least one serial program. Since we almost always need to coordinate the actions of multiple cores, writing parallel programs is almost always more complex than writing a serial program that solves the same problem. In fact, it is often *far* more complex. All the rules about careful design and development are usually *far* more important for the writing of parallel programs than they are for serial programs.

1.9 TYPOGRAPHICAL CONVENTIONS

We'll make use of the following typefaces in the text:

- Program text, displayed or within running text, will use the following typefaces:

```
/*  This is a short program */
#include <stdio.h>

int main(int argc, char* argv[]) {
   printf("hello, world\n");

   return 0;
}
```

- Definitions are given in the body of the text, and the term being defined is printed in boldface type: A **parallel program** can make use of multiple cores.
- When we need to refer to the environment in which a program is being developed, we'll assume that we're using a UNIX shell, and we'll use a `$` to indicate the shell prompt:

```
$ gcc –g –Wall –o hello hello.c
```

- We'll specify the syntax of function calls with fixed argument lists by including a sample argument list. For example, the integer absolute value function, `abs`, in `stdlib` might have its syntax specified with

```
int abs(int x);  /* Returns absolute value of int x */
```

  For more complicated syntax, we'll enclose required content in angle brackets `<>` and optional content in square brackets `[]`. For example, the C `if` statement might have its syntax specified as follows:

```
if ( <expression> )
   <statement1>
[else
   <statement2>]
```

  This says that the `if` statement must include an expression enclosed in parentheses, and the right parenthesis must be followed by a statement. This statement can be followed by an optional `else` clause. If the `else` clause is present, it must include a second statement.

1.10 SUMMARY

For many years we've enjoyed the fruits of ever faster processors. However, because of physical limitations the rate of performance improvement in conventional processors is decreasing. In order to increase the power of processors, chipmakers have turned to **multicore** integrated circuits, that is, integrated circuits with multiple conventional processors on a single chip.

Ordinary **serial** programs, which are programs written for a conventional single-core processor, usually cannot exploit the presence of multiple cores, and it's unlikely that translation programs will be able to shoulder all the work of **parallelizing** serial programs, meaning converting them into **parallel programs**, which can make use of multiple cores. As software developers, we need to learn to write parallel programs.

When we write parallel programs, we usually need to **coordinate** the work of the cores. This can involve **communication** among the cores, **load balancing**, and **synchronization** of the cores.

In this book we'll be learning to program parallel systems so that we can maximize their performance. We'll be using the C language with either MPI, Pthreads, or OpenMP. MPI is used for programming **distributed-memory** systems, and Pthreads and OpenMP are used for programming **shared-memory** systems. In distributed-memory systems, the cores have their own private memories, while in shared-memory systems, it's possible, in principle, for each core to access each memory location.

Concurrent programs can have multiple tasks in progress at any instant. **Parallel** and **distributed** programs usually have tasks that execute simultaneously. There isn't a hard and fast distinction between parallel and distributed, although in parallel programs, the tasks are usually more tightly coupled.

Parallel programs are usually very complex. So it's even more important to use good program development techniques with parallel programs.

1.11 EXERCISES

1.1 Devise formulas for the functions that calculate `my_first_i` and `my_last_i` in the global sum example. Remember that each core should be assigned roughly the same number of elements of computations in the loop. *Hint*: First consider the case when n is evenly divisible by p.

1.2 We've implicitly assumed that each call to `Compute_next_value` requires roughly the same amount of work as the other calls. How would you change your answer to the preceding question if call $i = k$ requires $k + 1$ times as much work as the call with $i = 0$? So if the first call ($i = 0$) requires 2 milliseconds, the second call ($i = 1$) requires 4, the third ($i = 2$) requires 6, and so on.

1.3 Try to write pseudo-code for the tree-structured global sum illustrated in Figure 1.1. Assume the number of cores is a power of two (1, 2, 4, 8, ...).

Hints: Use a variable `divisor` to determine whether a core should send its sum or receive and add. The `divisor` should start with the value 2 and be doubled after each iteration. Also use a variable `core_difference` to determine which core should be partnered with the current core. It should start with the value 1 and also be doubled after each iteration. For example, in the first iteration `0 % divisor = 0` and `1 % divisor = 1`, so 0 receives and adds, while 1 sends. Also in the first iteration `0 + core_difference = 1` and `1 – core_difference = 0`, so 0 and 1 are paired in the first iteration.

1.4 As an alternative to the approach outlined in the preceding problem, we can use C's bitwise operators to implement the tree-structured global sum. In order to see how this works, it helps to write down the binary (base 2) representation of each of the core ranks, and note the pairings during each stage:

	Stages		
Cores	*1*	*2*	*3*
$0_{10} = 000_2$	$1_{10} = 001_2$	$2_{10} = 010_2$	$4_{10} = 100_2$
$1_{10} = 001_2$	$0_{10} = 000_2$	×	×
$2_{10} = 010_2$	$3_{10} = 011_2$	$0_{10} = 000_2$	×
$3_{10} = 011_2$	$2_{10} = 010_2$	×	×
$4_{10} = 100_2$	$5_{10} = 101_2$	$6_{10} = 110_2$	$0_{10} = 000_2$
$5_{10} = 101_2$	$4_{10} = 100_2$	×	×
$6_{10} = 110_2$	$7_{10} = 111_2$	$4_{10} = 100_2$	×
$7_{10} = 111_2$	$6_{10} = 110_2$	×	×

From the table we see that during the first stage each core is paired with the core whose rank differs in the rightmost or first bit. During the second stage cores that continue are paired with the core whose rank differs in the second bit, and during the third stage cores are paired with the core whose rank differs in the third bit. Thus, if we have a binary value `bitmask` that is 001_2 for the first stage, 010_2 for the second, and 100_2 for the third, we can get the rank of the core we're paired with by "inverting" the bit in our rank that is nonzero in `bitmask`. This can be done using the bitwise exclusive or $\wedge$ operator.

Implement this algorithm in pseudo-code using the bitwise exclusive or and the left-shift operator.

1.5 What happens if your pseudo-code in Exercise 1.3 or Exercise 1.4 is run when the number of cores is *not* a power of two (e.g., 3, 5, 6, 7)? Can you modify the pseudo-code so that it will work correctly regardless of the number of cores?

1.6 Derive formulas for the number of receives and additions that core 0 carries out using

a. the original pseudo-code for a global sum, and

b. the tree-structured global sum.

Make a table showing the numbers of receives and additions carried out by core 0 when the two sums are used with $2, 4, 8, \ldots, 1024$ cores.

1.7 The first part of the global sum example—when each core adds its assigned computed values—is usually considered to be an example of data-parallelism, while the second part of the first global sum—when the cores send their partial sums to the master core, which adds them—could be considered to be an example of task-parallelism. What about the second part of the second global sum—when the cores use a tree structure to add their partial sums? Is this an example of data- or task-parallelism? Why?

1.8 Suppose the faculty are going to have a party for the students in the department.

- **a.** Identify tasks that can be assigned to the faculty members that will allow them to use task-parallelism when they prepare for the party. Work out a schedule that shows when the various tasks can be performed.
- **b.** We might hope that one of the tasks in the preceding part is cleaning the house where the party will be held. How can we use data-parallelism to partition the work of cleaning the house among the faculty?
- **c.** Use a combination of task- and data-parallelism to prepare for the party. (If there's too much work for the faculty, you can use TAs to pick up the slack.)

1.9 Write an essay describing a research problem in your major that would benefit from the use of parallel computing. Provide a rough outline of how parallelism would be used. Would you use task- or data-parallelism?

CHAPTER

2 Parallel Hardware and Parallel Software

It's perfectly feasible for specialists in disciplines other than computer science and computer engineering to write parallel programs. However, in order to write *efficient* parallel programs, we do need some knowledge of the underlying hardware and system software. It's also very useful to have some knowledge of different types of parallel software, so in this chapter we'll take a brief look at a few topics in hardware and software. We'll also take a brief look at evaluating program performance and a method for developing parallel programs. We'll close with a discussion of what kind of environment we might expect to be working in, and a few rules and assumptions we'll make in the rest of the book.

This is a long, broad chapter, so it may be a good idea to skim through some of the sections on a first reading so that you have a good idea of what's in the chapter. Then, when a concept or term in a later chapter isn't quite clear, it may be helpful to refer back to this chapter. In particular, you may want to skim over most of the material in "Modifications to the von Neumann Model," except "The Basics of Caching." Also, in the "Parallel Hardware" section, you can safely skim the material on "SIMD Systems" and "Interconnection Networks."

2.1 SOME BACKGROUND

Parallel hardware and software have grown out of conventional **serial** hardware and software: hardware and software that runs (more or less) a single job at a time. So in order to better understand the current state of parallel systems, let's begin with a brief look at a few aspects of serial systems.

2.1.1 The von Neumann architecture

The "classical" **von Neumann architecture** consists of **main memory,** a **central-processing unit** (CPU) or **processor** or **core,** and an **interconnection** between the memory and the CPU. Main memory consists of a collection of locations, each of which is capable of storing both instructions and data. Every location consists of an address, which is used to access the location and the contents of the location—the instructions or data stored in the location.

An Introduction to Parallel Programming. DOI: 10.1016/B978-0-12-374260-5.00002-6

The central processing unit is divided into a control unit and an arithmetic and logic unit (ALU). The control unit is responsible for deciding which instructions in a program should be executed, and the ALU is responsible for executing the actual instructions. Data in the CPU and information about the state of an executing program are stored in special, very fast storage called **registers**. The control unit has a special register called the **program counter**. It stores the address of the next instruction to be executed.

Instructions and data are transferred between the CPU and memory via the interconnect. This has traditionally been a **bus**, which consists of a collection of parallel wires and some hardware controlling access to the wires. A von Neumann machine executes a single instruction at a time, and each instruction operates on only a few pieces of data. See Figure 2.1.

When data or instructions are transferred from memory to the CPU, we sometimes say the data or instructions are **fetched** or **read** from memory. When data are transferred from the CPU to memory, we sometimes say the data are **written to memory** or **stored**. The separation of memory and CPU is often called the **von Neumann**

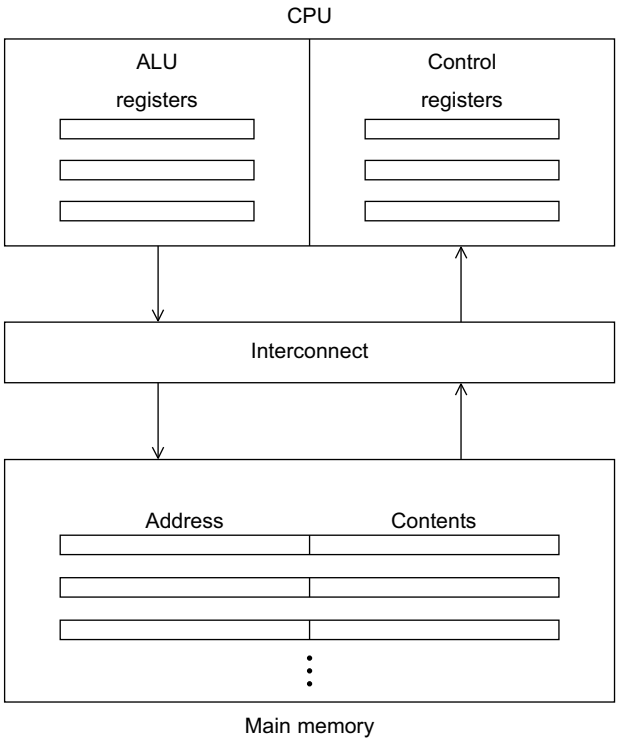


FIGURE 2.1

The von Neumann architecture

bottleneck, since the interconnect determines the rate at which instructions and data can be accessed. The potentially vast quantity of data and instructions needed to run a program is effectively isolated from the CPU. In 2010 CPUs are capable of executing instructions more than one hundred times faster than they can fetch items from main memory.

In order to better understand this problem, imagine that a large company has a single factory (the CPU) in one town and a single warehouse (main memory) in another. Further imagine that there is a single two-lane road joining the warehouse and the factory. All the raw materials used in manufacturing the products are stored in the warehouse. Also, all the finished products are stored in the warehouse before being shipped to customers. If the rate at which products can be manufactured is much larger than the rate at which raw materials and finished products can be transported, then it's likely that there will be a huge traffic jam on the road, and the employees and machinery in the factory will either be idle for extended periods or they will have to reduce the rate at which they produce finished products.

In order to address the von Neumann bottleneck, and, more generally, improve CPU performance, computer engineers and computer scientists have experimented with many modifications to the basic von Neumann architecture. Before discussing some of these modifications, let's first take a moment to discuss some aspects of the software that are used in both von Neumann systems and more modern systems.

2.1.2 Processes, multitasking, and threads

Recall that the **operating system**, or OS, is a major piece of software whose purpose is to manage hardware and software resources on a computer. It determines which programs can run and when they can run. It also controls the allocation of memory to running programs and access to peripheral devices such as hard disks and network interface cards.

When a user runs a program, the operating system creates a **process**—an instance of a computer program that is being executed. A process consists of several entities:

- The executable machine language program.
- A block of memory, which will include the executable code, a **call stack** that keeps track of active functions, a **heap**, and some other memory locations.
- Descriptors of resources that the operating system has allocated to the process—for example, file descriptors.
- Security information—for example, information specifying which hardware and software resources the process can access.
- Information about the state of the process, such as whether the process is ready to run or is waiting on some resource, the content of the registers, and information about the process' memory.

Most modern operating systems are **multitasking**. This means that the operating system provides support for the apparent simultaneous execution of multiple programs. This is possible even on a system with a single core, since each process runs

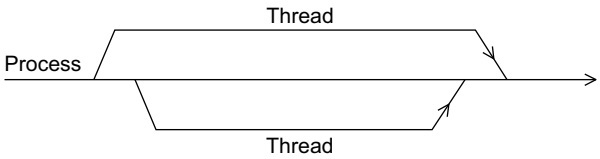


FIGURE 2.2

A process and two threads

for a small interval of time (typically a few milliseconds), often called a **time slice**. After one running program has executed for a time slice, the operating system can run a different program. A multitasking OS may change the running process many times a minute, even though changing the running process can take a long time.

In a multitasking OS if a process needs to wait for a resource—for example, it needs to read data from external storage—it will **block**. This means that it will stop executing and the operating system can run another process. However, many programs can continue to do useful work even though the part of the program that is currently executing must wait on a resource. For example, an airline reservation system that is blocked waiting for a seat map for one user could provide a list of available flights to another user. **Threading** provides a mechanism for programmers to divide their programs into more or less independent tasks with the property that when one thread is blocked another thread can be run. Furthermore, in most systems it's possible to switch between threads much faster than it's possible to switch between processes. This is because threads are "lighter weight" than processes. Threads are contained within processes, so they can use the same executable, and they usually share the same memory and the same I/O devices. In fact, two threads belonging to one process can share most of the process' resources. The two most important exceptions are that they'll need a record of their own program counters and they'll need their own call stacks so that they can execute independently of each other.

If a process is the "master" thread of execution and threads are started and stopped by the process, then we can envision the process and its subsidiary threads as lines: when a thread is started, it **forks** off the process; when a thread terminates, it **joins** the process. See Figure 2.2.

2.2 MODIFICATIONS TO THE VON NEUMANN MODEL

As we noted earlier, since the first electronic digital computers were developed back in the 1940s, computer scientists and computer engineers have made many improvements to the basic von Neumann architecture. Many are targeted at reducing the problem of the von Neumann bottleneck, but many are also targeted at simply making CPUs faster. In this section we'll look at three of these improvements: caching, virtual memory, and low-level parallelism.

2.2.1 The basics of caching

Caching is one of the most widely used methods of addressing the von Neumann bottleneck. To understand the ideas behind caching, recall our example. A company has a factory (CPU) in one town and a warehouse (main memory) in another, and there is a single, two-lane road joining the factory and the warehouse. There are a number of possible solutions to the problem of transporting raw materials and finished products between the warehouse and the factory. One is to widen the road. Another is to move the factory and/or the warehouse or to build a unified factory and warehouse. Caching exploits both of these ideas. Rather than transporting a single instruction or data item, we can use an effectively wider interconnection, an interconnection that can transport more data or more instructions in a single memory access. Also, rather than storing all data and instructions exclusively in main memory, we can store blocks of data and instructions in special memory that is effectively closer to the registers in the CPU.

In general a **cache** is a collection of memory locations that can be accessed in less time than some other memory locations. In our setting, when we talk about caches we'll usually mean a **CPU cache**, which is a collection of memory locations that the CPU can access more quickly than it can access main memory. A CPU cache can either be located on the same chip as the CPU or it can be located on a separate chip that can be accessed much faster than an ordinary memory chip.

Once we have a cache, an obvious problem is deciding which data and instructions should be stored in the cache. The universally used principle is based on the idea that programs tend to use data and instructions that are physically close to recently used data and instructions. After executing an instruction, programs typically execute the next instruction; branching tends to be relatively rare. Similarly, after a program has accessed one memory location, it often accesses a memory location that is physically nearby. An extreme example of this is in the use of arrays. Consider the loop

```
float z[1000];
. . .
sum = 0.0;
for (i = 0; i < 1000; i++)
   sum += z[i];
```

Arrays are allocated as blocks of contiguous memory locations. So, for example, the location storing `z[1]` immediately follows the location `z[0]`. Thus as long as `i < 999`, the read of `z[i]` is immediately followed by a read of `z[i+1]`.

The principle that an access of one location is followed by an access of a nearby location is often called **locality**. After accessing one memory location (instruction or data), a program will typically access a nearby location (**spatial** locality) in the near future (**temporal** locality).

In order to exploit the principle of locality, the system uses an effectively *wider* interconnect to access data and instructions. That is, a memory access will effectively operate on blocks of data and instructions instead of individual instructions and individual data items. These blocks are called **cache blocks** or **cache lines**. A typical cache line stores 8 to 16 times as much information as a single memory

location. In our example, if a cache line stores 16 floats, then when we first go to add `sum += z[0]`, the system might read the first 16 elements of `z`, `z[0]`, `z[1]`, ..., `z[15]` from memory into cache. So the next 15 additions will use elements of `z` that are already in the cache.

Conceptually, it's often convenient to think of a CPU cache as a single monolithic structure. In practice, though, the cache is usually divided into **levels**: the first level (L1) is the smallest and the fastest, and higher levels (L2, L3, ...) are larger and slower. Most systems currently, in 2010, have at least two levels and having three levels is quite common. Caches usually store copies of information in slower memory, and, if we think of a lower-level (faster, smaller) cache as a cache for a higher level, this usually applies. So, for example, a variable stored in a level 1 cache will also be stored in level 2. However, some multilevel caches don't duplicate information that's available in another level. For these caches, a variable in a level 1 cache might not be stored in any other level of the cache, but it *would* be stored in main memory.

When the CPU needs to access an instruction or data, it works its way down the cache hierarchy: First it checks the level 1 cache, then the level 2, and so on. Finally, if the information needed isn't in any of the caches, it accesses main memory. When a cache is checked for information and the information is available, it's called a **cache hit** or just a **hit**. If the information isn't available, it's called a **cache miss** or a **miss**. Hit or miss is often modified by the level. For example, when the CPU attempts to access a variable, it might have an L1 miss and an L2 hit.

Note that the memory access terms **read** and **write** are also used for caches. For example, we might read an instruction from an L2 cache, and we might write data to an L1 cache.

When the CPU attempts to read data or instructions and there's a cache read-miss, it will read from memory the cache line that contains the needed information and store it in the cache. This may stall the processor, while it waits for the slower memory: the processor may stop executing statements from the current program until the required data or instructions have been fetched from memory. So in our example, when we read `z[0]`, the processor may stall while the cache line containing `z[0]` is transferred from memory into the cache.

When the CPU writes data to a cache, the value in the cache and the value in main memory are different or **inconsistent**. There are two basic approaches to dealing with the inconsistency. In **write-through** caches, the line is written to main memory when it is written to the cache. In **write-back** caches, the data isn't written immediately. Rather, the updated data in the cache is marked *dirty,* and when the cache line is replaced by a new cache line from memory, the dirty line is written to memory.

2.2.2 Cache mappings

Another issue in cache design is deciding where lines should be stored. That is, if we fetch a cache line from main memory, where in the cache should it be placed? The answer to this question varies from system to system. At one extreme is a **fully associative** cache, in which a new line can be placed at any location in the cache. At

the other extreme is a **direct mapped** cache, in which each cache line has a unique location in the cache to which it will be assigned. Intermediate schemes are called ***n*-way set associative**. In these schemes, each cache line can be placed in one of *n* different locations in the cache. For example, in a two way set associative cache, each line can be mapped to one of two locations.

As an example, suppose our main memory consists of 16 lines with indexes 0–15, and our cache consists of 4 lines with indexes 0–3. In a fully associative cache, line 0 can be assigned to cache location 0, 1, 2, or 3. In a direct mapped cache, we might assign lines by looking at their remainder after division by 4. So lines 0, 4, 8, and 12 would be mapped to cache index 0, lines 1, 5, 9, and 13 would be mapped to cache index 1, and so on. In a two way set associative cache, we might group the cache into two sets: indexes 0 and 1 form one set—set 0—and indexes 2 and 3 form another—set 1. So we could use the remainder of the main memory index modulo 2, and cache line 0 would be mapped to either cache index 0 or cache index 1. See Table 2.1.

When more than one line in memory can be mapped to several different locations in a cache (fully associative and *n*-way set associative), we also need to be able to decide which line should be replaced or **evicted**. In our preceding example, if, for example, line 0 is in location 0 and line 2 is in location 1, where would we store line 4? The most commonly used scheme is called **least recently used**. As the name suggests, the cache has a record of the relative order in which the blocks have been

Table 2.1 Assignments of a 16-line Main Memory to a 4-line Cache

	Cache Location		
Memory Index	*Fully Assoc*	*Direct Mapped*	*2-way*
0	0, 1, 2, or 3	0	0 or 1
1	0, 1, 2, or 3	1	2 or 3
2	0, 1, 2, or 3	2	0 or 1
3	0, 1, 2, or 3	3	2 or 3
4	0, 1, 2, or 3	0	0 or 1
5	0, 1, 2, or 3	1	2 or 3
6	0, 1, 2, or 3	2	0 or 1
7	0, 1, 2, or 3	3	2 or 3
8	0, 1, 2, or 3	0	0 or 1
9	0, 1, 2, or 3	1	2 or 3
10	0, 1, 2, or 3	2	0 or 1
11	0, 1, 2, or 3	3	2 or 3
12	0, 1, 2, or 3	0	0 or 1
13	0, 1, 2, or 3	1	2 or 3
14	0, 1, 2, or 3	2	0 or 1
15	0, 1, 2, or 3	3	2 or 3

used, and if line 0 were used more recently than line 2, then line 2 would be evicted and replaced by line 4.

2.2.3 Caches and programs: an example

It's important to remember that the workings of the CPU cache are controlled by the system hardware, and we, the programmers, don't directly determine which data and which instructions are in the cache. However, knowing the principle of spatial and temporal locality allows us to have some indirect control over caching. As an example, C stores two-dimensional arrays in "row-major" order. That is, although we think of a two-dimensional array as a rectangular block, memory is effectively a huge one-dimensional array. So in row-major storage, we store row 0 first, then row 1, and so on. In the following two code segments, we would expect the first pair of nested loops to have much better performance than the second, since it's accessing the data in the two-dimensional array in contiguous blocks.

```
double A[MAX][MAX], x[MAX], y[MAX];
. . .
/* Initialize A and x, assign y = 0 */
. . .
/* First pair of loops */
for (i = 0; i < MAX; i++)
   for (j = 0; j < MAX; j++)
      y[i] += A[i][j]*x[j];
. . .
/* Assign y = 0 */
. . .
/* Second pair of loops */
for (j = 0; j < MAX; j++)
   for (i = 0; i < MAX; i++)
      y[i] += A[i][j]*x[j];
```

To better understand this, suppose `MAX` is four, and the elements of `A` are stored in memory as follows:

Cache Line	Elements of A			
0	A[0][0]	A[0][1]	A[0][2]	A[0][3]
1	A[1][0]	A[1][1]	A[1][2]	A[1][3]
2	A[2][0]	A[2][1]	A[2][2]	A[2][3]
3	A[3][0]	A[3][1]	A[3][2]	A[3][3]

So, for example, `A[0][1]` is stored immediately after `A[0][0]` and `A[1][0]` is stored immediately after `A[0][3]`.

Let's suppose that none of the elements of `A` are in the cache when each pair of loops starts executing. Let's also suppose that a cache line consists of four elements

of A and A[0][0] is the first element of a cache line. Finally, we'll suppose that the cache is direct mapped and it can only store eight elements of A, or two cache lines. (We won't worry about x and y.)

Both pairs of loops attempt to first access A[0][0]. Since it's not in the cache, this will result in a cache miss, and the system will read the line consisting of the first row of A, A[0][0], A[0][1], A[0][2], A[0][3], into the cache. The first pair of loops then accesses A[0][1], A[0][2], A[0][3], all of which are in the cache, and the next miss in the first pair of loops will occur when the code accesses A[1][0]. Continuing in this fashion, we see that the first pair of loops will result in a total of four misses when it accesses elements of A, one for each row. Note that since our hypothetical cache can only store two lines or eight elements of A, when we read the first element of row two and the first element of row three, one of the lines that's already in the cache will have to be evicted from the cache, but once a line is evicted, the first pair of loops won't need to access the elements of that line again.

After reading the first row into the cache, the second pair of loops needs to then access A[1][0], A[2][0], A[3][0], none of which are in the cache. So the next three accesses of A will also result in misses. Furthermore, because the cache is small, the reads of A[2][0] and A[3][0] will require that lines already in the cache be evicted. Since A[2][0] is stored in cache line 2, reading its line will evict line 0, and reading A[3][0] will evict line 1. After finishing the first pass through the outer loop, we'll next need to access A[0][1], which was evicted with the rest of the first row. So we see that *every* time we read an element of A, we'll have a miss, and the second pair of loops results in 16 misses.

Thus, we'd expect the first pair of nested loops to be much faster than the second. In fact, if we run the code on one of our systems with MAX = 1000, the first pair of nested loops is approximately three times faster than the second pair.

2.2.4 Virtual memory

Caches make it possible for the CPU to quickly access instructions and data that are in main memory. However, if we run a very large program or a program that accesses very large data sets, all of the instructions and data may not fit into main memory. This is especially true with multitasking operating systems; in order to switch between programs and create the illusion that multiple programs are running simultaneously, the instructions and data that will be used during the next time slice should be in main memory. Thus, in a multitasking system, even if the main memory is very large, many running programs must share the available main memory. Furthermore, this sharing must be done in such a way that each program's data and instructions are protected from corruption by other programs.

Virtual memory was developed so that main memory can function as a cache for secondary storage. It exploits the principle of spatial and temporal locality by keeping in main memory only the active parts of the many running programs; those

parts that are idle are kept in a block of secondary storage called **swap space**. Like CPU caches, virtual memory operates on blocks of data and instructions. These blocks are commonly called **pages**, and since secondary storage access can be hundreds of thousands of times slower than main memory access, pages are relatively large—most systems have a fixed page size that currently ranges from 4 to 16 kilobytes.

We may run into trouble if we try to assign physical memory addresses to pages when we compile a program. If we do this, then each page of the program can only be assigned to one block of memory, and with a multitasking operating system, we're likely to have many programs wanting to use the same block of memory. In order to avoid this problem, when a program is compiled, its pages are assigned *virtual* page numbers. Then, when the program is run, a table is created that maps the virtual page numbers to physical addresses. When the program is run and it refers to a virtual address, this **page table** is used to translate the virtual address into a physical address. If the creation of the page table is managed by the operating system, it can ensure that the memory used by one program doesn't overlap the memory used by another.

A drawback to the use of a page table is that it can double the time needed to access a location in main memory. Suppose, for example, that we want to execute an instruction in main memory. Then our executing program will have the *virtual* address of this instruction, but before we can find the instruction in memory, we'll need to translate the virtual address into a physical address. In order to do this, we'll need to find the page in memory that contains the instruction. Now the virtual page number is stored as a part of the virtual address. As an example, suppose our addresses are 32 bits and our pages are 4 kilobytes = 4096 bytes. Then each byte in the page can be identified with 12 bits, since $2^{12} = 4096$. Thus, we can use the low-order 12 bits of the virtual address to locate a byte within a page, and the remaining bits of the virtual address can be used to locate an individual page. See Table 2.2. Observe that the virtual page number can be computed from the virtual address without going to memory. However, once we've found the virtual page number, we'll need to access the page table to translate it into a physical page. If the required part of the page table isn't in cache, we'll need to load it from memory. After it's loaded, we can translate our virtual address to a physical address and get the required instruction.

Table 2.2 Virtual Address Divided into Virtual Page Number and Byte Offset

Virtual Address									
Virtual Page Number					Byte Offset				
31	30	⋯	13	12	11	10	⋯	1	0
1	0	⋯	1	1	0	0	⋯	1	1

This is clearly a problem. Although multiple programs can use main memory at more or less the same time, using a page table has the potential to significantly increase each program's overall run-time. In order to address this issue, processors have a special address translation cache called a **translation-lookaside buffer**, or TLB. It caches a small number of entries (typically 16–512) from the page table in very fast memory. Using the principle of spatial and temporal locality, we would expect that most of our memory references will be to pages whose physical address is stored in the TLB, and the number of memory references that require accesses to the page table in main memory will be substantially reduced.

The terminology for the TLB is the same as the terminology for caches. When we look for an address and the virtual page number is in the TLB, it's called a TLB *hit.* If it's not in the TLB, it's called a *miss.* The terminology for the page table, however, has an important difference from the terminology for caches. If we attempt to access a page that's not in memory, that is, the page table doesn't have a valid physical address for the page and the page is only stored on disk, then the attempted access is called a **page fault**.

The relative slowness of disk accesses has a couple of additional consequences for virtual memory. First, with CPU caches we could handle write-misses with either a write-through or write-back scheme. With virtual memory, however, disk accesses are so expensive that they should be avoided whenever possible, so virtual memory always uses a write-back scheme. This can be handled by keeping a bit on each page in memory that indicates whether the page has been updated. If it has been updated, when it is evicted from main memory, it will be written to disk. Second, since disk accesses are so slow, management of the page table and the handling of disk accesses can be done by the operating system. Thus, even though we as programmers don't directly control virtual memory, unlike CPU caches, which are handled by system hardware, virtual memory is usually controlled by a combination of system hardware and operating system software.

2.2.5 Instruction-level parallelism

Instruction-level parallelism, or ILP, attempts to improve processor performance by having multiple processor components or **functional units** simultaneously executing instructions. There are two main approaches to ILP: **pipelining**, in which functional units are arranged in stages, and **multiple issue**, in which multiple instructions can be simultaneously initiated. Both approaches are used in virtually all modern CPUs.

Pipelining

The principle of pipelining is similar to a factory assembly line: while one team is bolting a car's engine to the chassis, another team can connect the transmission to the engine and the driveshaft of a car that's already been processed by the first team, and a third team can bolt the body to the chassis in a car that's been processed by

the first two teams. As an example involving computation, suppose we want to add the floating point numbers 9.87×10^4 and 6.54×10^3. Then we can use the following steps:

Time	Operation	Operand 1	Operand 2	Result
0	Fetch operands	9.87×10^4	6.54×10^3	
1	Compare exponents	9.87×10^4	6.54×10^3	
2	Shift one operand	9.87×10^4	0.654×10^4	
3	Add	9.87×10^4	0.654×10^4	10.524×10^4
4	Normalize result	9.87×10^4	0.654×10^4	1.0524×10^5
5	Round result	9.87×10^4	0.654×10^4	1.05×10^5
6	Store result	9.87×10^4	0.654×10^4	1.05×10^5

Here we're using base 10 and a three digit mantissa or significand with one digit to the left of the decimal point. Thus, in the example, normalizing shifts the decimal point one unit to the left, and rounding rounds to three digits.

Now if each of the operations takes one nanosecond (10^{-9} seconds), the addition operation will take seven nanoseconds. So if we execute the code

```
float x[1000], y[1000], z[1000];
. . .
for (i = 0; i < 1000; i++)
   z[i] = x[i] + y[i];
```

the `for` loop will take something like 7000 nanoseconds.

As an alternative, suppose we divide our floating point adder into seven separate pieces of hardware or functional units. The first unit will fetch two operands, the second will compare exponents, and so on. Also suppose that the output of one functional unit is the input to the next. So, for example, the output of the functional unit that adds the two values is the input to the unit that normalizes the result. Then a single floating point addition will still take seven nanoseconds. However, when we execute the `for` loop, we can fetch `x[1]` and `y[1]` while we're comparing the exponents of `x[0]` and `y[0]`. More generally, it's possible for us to simultaneously execute seven different stages in seven different additions. See Table 2.3. From the table we see that after time 5, the pipelined loop produces a result every nanosecond, instead of every seven nanoseconds, so the total time to execute the `for` loop has been reduced from 7000 nanoseconds to 1006 nanoseconds—an improvement of almost a factor of seven.

In general, a pipeline with k stages won't get a k-fold improvement in performance. For example, if the times required by the various functional units are different, then the stages will effectively run at the speed of the slowest functional unit. Furthermore, delays such as waiting for an operand to become available can cause the pipeline to stall. See Exercise 2.1 for more details on the performance of pipelines.

Table 2.3 Pipelined Addition. Numbers in the Table Are Subscripts of Operands/Results

Time	Fetch	Compare	Shift	Add	Normalize	Round	Store
0	0						
1	1	0					
2	2	1	0				
3	3	2	1	0			
4	4	3	2	1	0		
5	5	4	3	2	1	0	
6	6	5	4	3	2	1	0
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
999	999	998	997	996	995	994	993
1000		999	998	997	996	995	994
1001			999	998	997	996	995
1002				999	998	997	996
1003					999	998	997
1004						999	998
1005							999

Multiple issue

Pipelines improve performance by taking individual pieces of hardware or functional units and connecting them in sequence. Multiple issue processors replicate functional units and try to simultaneously execute different instructions in a program. For example, if we have two complete floating point adders, we can approximately halve the time it takes to execute the loop

```
for (i = 0; i < 1000; i++)
   z[i] = x[i] + y[i];
```

While the first adder is computing `z[0]`, the second can compute `z[1]`; while the first is computing `z[2]`, the second can compute `z[3]`; and so on.

If the functional units are scheduled at compile time, the multiple issue system is said to use **static** multiple issue. If they're scheduled at run-time, the system is said to use **dynamic** multiple issue. A processor that supports dynamic multiple issue is sometimes said to be **superscalar**.

Of course, in order to make use of multiple issue, the system must find instructions that can be executed simultaneously. One of the most important techniques is **speculation**. In speculation, the compiler or the processor makes a guess about an instruction, and then executes the instruction on the basis of the guess. As a simple example, in the following code, the system might predict that the outcome of `z = x + y` will give `z` a positive value, and, as a consequence, it will assign `w = x`.

```
z = x + y;
if (z > 0)
   w = x;
else
   w = y;
```

As another example, in the code

```
z = x + y;
w = *a_p;  /* a_p is a pointer */
```

the system might predict that `a_p` does not refer to `z`, and hence it can simultaneously execute the two assignments.

As both examples make clear, speculative execution must allow for the possibility that the predicted behavior is incorrect. In the first example, we will need to go back and execute the assignment `w = y` if the assignment `z = x + y` results in a value that's not positive. In the second example, if `a_p` does point to `z`, we'll need to re-execute the assignment `w = *a_p`.

If the compiler does the speculation, it will usually insert code that tests whether the speculation was correct, and, if not, takes corrective action. If the hardware does the speculation, the processor usually stores the result(s) of the speculative execution in a buffer. When it's known that the speculation was correct, the contents of the buffer are transferred to registers or memory. If the speculation was incorrect, the contents of the buffer are discarded and the instruction is re-executed.

While dynamic multiple issue systems can execute instructions out of order, in current generation systems the instructions are still loaded in order and the results of the instructions are also committed in order. That is, the results of instructions are written to registers and memory in the program-specified order.

Optimizing compilers, on the other hand, can reorder instructions. This, as we'll see later, can have important consequences for shared-memory programming.

2.2.6 Hardware multithreading

ILP can be very difficult to exploit: it is a program with a long sequence of dependent statements offers few opportunities. For example, in a direct calculation of the Fibonacci numbers

```
f[0] = f[1] = 1;
for (i = 2; i <= n; i++)
   f[i] = f[i-1] + f[i-2];
```

there's essentially no opportunity for simultaneous execution of instructions.

Thread-level parallelism, or TLP, attempts to provide parallelism through the simultaneous execution of different threads, so it provides a **coarser-grained** parallelism than ILP, that is, the program units that are being simultaneously executed—threads—are larger or coarser than the **finer-grained** units—individual instructions.

Hardware multithreading provides a means for systems to continue doing useful work when the task being currently executed has stalled—for example, if the current task has to wait for data to be loaded from memory. Instead of looking for parallelism in the currently executing thread, it may make sense to simply run another thread. Of course, in order for this to be useful, the system must support very rapid switching between threads. For example, in some older systems, threads were simply implemented as processes, and in the time it took to switch between processes, thousands of instructions could be executed.

In **fine-grained** multithreading, the processor switches between threads after each instruction, skipping threads that are stalled. While this approach has the potential to avoid wasted machine time due to stalls, it has the drawback that a thread that's ready to execute a long sequence of instructions may have to wait to execute every instruction. **Coarse-grained** multithreading attempts to avoid this problem by only switching threads that are stalled waiting for a time-consuming operation to complete (e.g., a load from main memory). This has the virtue that switching threads doesn't need to be nearly instantaneous. However, the processor can be idled on shorter stalls, and thread switching will also cause delays.

Simultaneous multithreading, or SMT, is a variation on fine-grained multithreading. It attempts to exploit superscalar processors by allowing multiple threads to make use of the multiple functional units. If we designate "preferred" threads—threads that have many instructions ready to execute—we can somewhat reduce the problem of thread slowdown.

2.3 PARALLEL HARDWARE

Multiple issue and pipelining can clearly be considered to be parallel hardware, since functional units are replicated. However, since this form of parallelism isn't usually visible to the programmer, we're treating both of them as extensions to the basic von Neumann model, and for our purposes, parallel hardware will be limited to hardware that's visible to the programmer. In other words, if she can readily modify her source code to exploit it, or if she must modify her source code to exploit it, then we'll consider the hardware to be parallel.

2.3.1 SIMD systems

In parallel computing, **Flynn's taxonomy** [18] is frequently used to classify computer architectures. It classifies a system according to the number of instruction streams and the number of data streams it can simultaneously manage. A classical von Neumann system is therefore a **single instruction stream, single data stream**, or SISD system, since it executes a single instruction at a time and it can fetch or store one item of data at a time.

Single instruction, multiple data, or SIMD, systems are parallel systems. As the name suggests, SIMD systems operate on multiple data streams by applying the

same instruction to multiple data items, so an abstract SIMD system can be thought of as having a single control unit and multiple ALUs. An instruction is broadcast from the control unit to the ALUs, and each ALU either applies the instruction to the current data item, or it is idle. As an example, suppose we want to carry out a "vector addition." That is, suppose we have two arrays `x` and `y`, each with n elements, and we want to add the elements of `y` to the elements of `x`:

```
for (i = 0; i < n; i++)
   x[i] += y[i];
```

Suppose further that our SIMD system has n ALUs. Then we could load `x[i]` and `y[i]` into the ith ALU, have the ith ALU add `y[i]` to `x[i]`, and store the result in `x[i]`. If the system has m ALUs and $m < n$, we can simply execute the additions in blocks of m elements at a time. For example, if $m = 4$ and $n = 15$, we can first add elements 0 to 3, then elements 4 to 7, then elements 8 to 11, and finally elements 12 to 14. Note that in the last group of elements in our example—elements 12 to 14—we're only operating on three elements of `x` and `y`, so one of the four ALUs will be idle.

The requirement that all the ALUs execute the same instruction or are idle can seriously degrade the overall performance of a SIMD system. For example, suppose we only want to carry out the addition if `y[i]` is positive:

```
for (i = 0; i < n; i++)
   if (y[i] > 0.0) x[i] += y[i];
```

In this setting, we must load each element of `y` into an ALU and determine whether it's positive. If `y[i]` is positive, we can proceed to carry out the addition. Otherwise, the ALU storing `y[i]` will be idle while the other ALUs carry out the addition.

Note also that in a "classical" SIMD system, the ALUs must operate synchronously, that is, each ALU must wait for the next instruction to be broadcast before proceeding. Further, the ALUs have no instruction storage, so an ALU can't delay execution of an instruction by storing it for later execution.

Finally, as our first example shows, SIMD systems are ideal for parallelizing simple loops that operate on large arrays of data. Parallelism that's obtained by dividing data among the processors and having the processors all apply (more or less) the same instructions to their subsets of the data is called **data-parallelism**. SIMD parallelism can be very efficient on large data parallel problems, but SIMD systems often don't do very well on other types of parallel problems.

SIMD systems have had a somewhat checkered history. In the early 1990s a maker of SIMD systems (Thinking Machines) was the largest manufacturer of parallel supercomputers. However, by the late 1990s the only widely produced SIMD systems were **vector processors**. More recently, graphics processing units, or GPUs, and desktop CPUs are making use of aspects of SIMD computing.

Vector processors

Although what constitutes a vector processor has changed over the years, their key characteristic is that they can operate on arrays or *vectors* of data, while conventional

CPUs operate on individual data elements or *scalars.* Typical recent systems have the following characteristics:

- *Vector registers.* These are registers capable of storing a vector of operands and operating simultaneously on their contents. The vector length is fixed by the system, and can range from 4 to 128 64-bit elements.
- *Vectorized and pipelined functional units.* Note that the same operation is applied to each element in the vector, or, in the case of operations like addition, the same operation is applied to each pair of corresponding elements in the two vectors. Thus, vector operations are SIMD.
- *Vector instructions.* These are instructions that operate on vectors rather than scalars. If the vector length is `vector_length`, these instructions have the great virtue that a simple loop such as

  ```
  for (i = 0; i < n; i++)
     x[i] += y[i];
  ```

  requires only a single load, add, and store for each block of `vector_length` elements, while a conventional system requires a load, add, and store for each element.
- *Interleaved memory.* The memory system consists of multiple "banks" of memory, which can be accessed more or less independently. After accessing one bank, there will be a delay before it can be reaccessed, but a different bank can be accessed much sooner. So if the elements of a vector are distributed across multiple banks, there can be little to no delay in loading/storing successive elements.
- *Strided memory access and hardware scatter/gather.* In *strided memory access,* the program accesses elements of a vector located at fixed intervals. For example, accessing the first element, the fifth element, the ninth element, and so on, would be strided access with a stride of four. Scatter/gather (in this context) is writing (scatter) or reading (gather) elements of a vector located at irregular intervals—for example, accessing the first element, the second element, the fourth element, the eighth element, and so on. Typical vector systems provide special hardware to accelerate strided access and scatter/gather.

Vector processors have the virtue that for many applications, they are very fast and very easy to use. Vectorizing compilers are quite good at identifying code that can be vectorized. Further, they identify loops that cannot be vectorized, and they often provide information about why a loop couldn't be vectorized. The user can thereby make informed decisions about whether it's possible to rewrite the loop so that it will vectorize. Vector systems have very high memory bandwidth, and every data item that's loaded is actually used, unlike cache-based systems that may not make use of every item in a cache line. On the other hand, they don't handle irregular data structures as well as other parallel architectures, and there seems to be a very finite limit to their **scalability**, that is, their ability to handle ever larger problems. It's difficult to see how systems could be created that would operate on ever longer vectors. Current generation systems scale by increasing the number of vector processors, not the

vector length. Current commodity systems provide limited support for operations on very short vectors, while processors that operate on long vectors are custom manufactured, and, consequently, very expensive.

Graphics processing units

Real-time graphics application programming interfaces, or APIs, use points, lines, and triangles to internally represent the surface of an object. They use a **graphics processing pipeline** to convert the internal representation into an array of pixels that can be sent to a computer screen. Several of the stages of this pipeline are programmable. The behavior of the programmable stages is specified by functions called **shader functions.** The shader functions are typically quite short—often just a few lines of C code. They're also implicitly parallel, since they can be applied to multiple elements (e.g., vertices) in the graphics stream. Since the application of a shader function to nearby elements often results in the same flow of control, GPUs can optimize performance by using SIMD parallelism, and in the current generation all GPUs use SIMD parallelism. This is obtained by including a large number of ALUs (e.g., 80) on each GPU processing core.

Processing a single image can require very large amounts of data—hundreds of megabytes of data for a single image is not unusual. GPUs therefore need to maintain very high rates of data movement, and in order to avoid stalls on memory accesses, they rely heavily on hardware multithreading; some systems are capable of storing the state of more than a hundred suspended threads for each executing thread. The actual number of threads depends on the amount of resources (e.g., registers) needed by the shader function. A drawback here is that many threads processing a lot of data are needed to keep the ALUs busy, and GPUs may have relatively poor performance on small problems.

It should be stressed that GPUs are not pure SIMD systems. Although the ALUs on a given core do use SIMD parallelism, current generation GPUs can have dozens of cores, which are capable of executing independent instruction streams.

GPUs are becoming increasingly popular for general, high-performance computing, and several languages have been developed that allow users to exploit their power. For further details see [30].

2.3.2 MIMD systems

Multiple instruction, multiple data, or MIMD, systems support multiple simultaneous instruction streams operating on multiple data streams. Thus, MIMD systems typically consist of a collection of fully independent processing units or cores, each of which has its own control unit and its own ALU. Furthermore, unlike SIMD systems, MIMD systems are usually **asynchronous**, that is, the processors can operate at their own pace. In many MIMD systems there is no global clock, and there may be no relation between the system times on two different processors. In fact, unless the programmer imposes some synchronization, even if the processors are executing exactly the same sequence of instructions, at any given instant they may be executing different statements.

As we noted in Chapter 1, there are two principal types of MIMD systems: shared-memory systems and distributed-memory systems. In a **shared-memory system** a collection of autonomous processors is connected to a memory system via an interconnection network, and each processor can access each memory location. In a shared-memory system, the processors usually communicate implicitly by accessing shared data structures. In a **distributed-memory system**, each processor is paired with its own *private* memory, and the processor-memory pairs communicate over an interconnection network. So in distributed-memory systems the processors usually communicate explicitly by sending messages or by using special functions that provide access to the memory of another processor. See Figures 2.3 and 2.4.

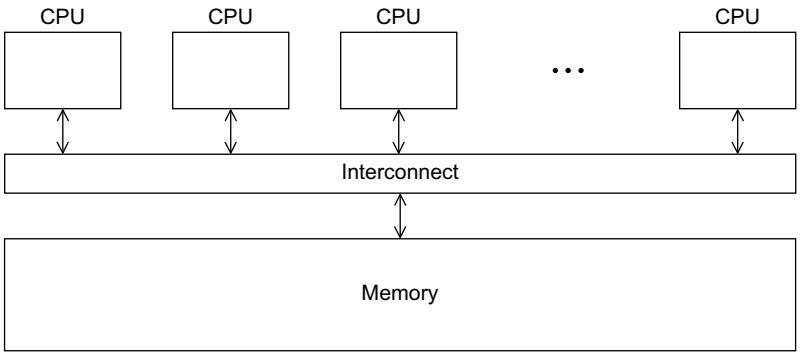


FIGURE 2.3

A shared-memory system

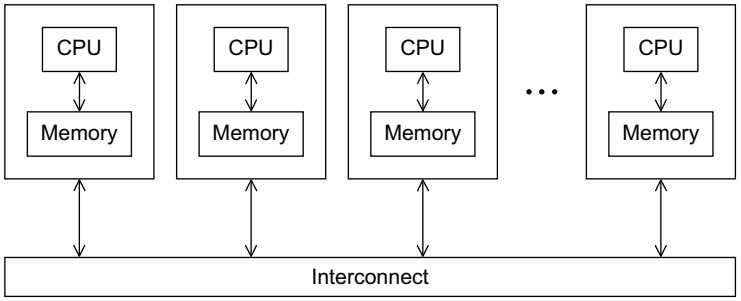


FIGURE 2.4

A distributed-memory system

Shared-memory systems

The most widely available shared-memory systems use one or more **multicore** processors. As we discussed in Chapter 1, a multicore processor has multiple CPUs or cores on a single chip. Typically, the cores have private level 1 caches, while other caches may or may not be shared between the cores.

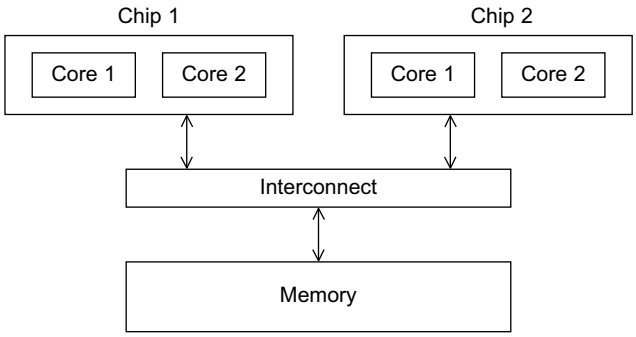


FIGURE 2.5

A UMA multicore system

In shared-memory systems with multiple multicore processors, the interconnect can either connect all the processors directly to main memory or each processor can have a direct connection to a block of main memory, and the processors can access each others' blocks of main memory through special hardware built into the processors. See Figures 2.5 and 2.6. In the first type of system, the time to access all the memory locations will be the same for all the cores, while in the second type a memory location to which a core is directly connected can be accessed more quickly than a memory location that must be accessed through another chip. Thus, the first type of system is called a **uniform memory access**, or UMA, system, while the second type is called a **nonuniform memory access**, or NUMA, system. UMA systems are usually easier to program, since the programmer doesn't need to worry about different access times for different memory locations. This advantage can be offset by the faster access to the directly connected memory in NUMA systems. Furthermore, NUMA systems have the potential to use larger amounts of memory than UMA systems.

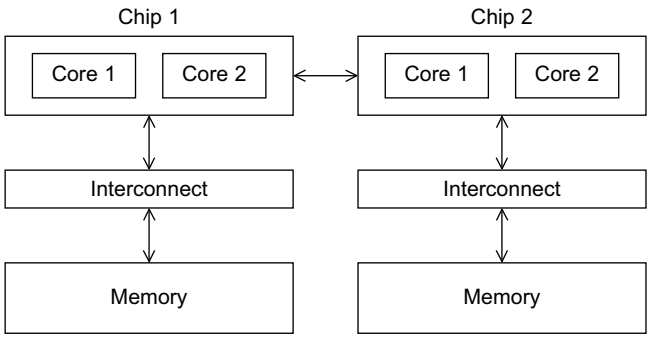


FIGURE 2.6

A NUMA multicore system

Distributed-memory systems

The most widely available distributed-memory systems are called **clusters**. They are composed of a collection of commodity systems—for example, PCs—connected by a commodity interconnection network—for example, Ethernet. In fact, the **nodes** of these systems, the individual computational units joined together by the communication network, are usually shared-memory systems with one or more multicore processors. To distinguish such systems from pure distributed-memory systems, they are sometimes called **hybrid systems**. Nowadays, it's usually understood that a cluster will have shared-memory nodes.

The **grid** provides the infrastructure necessary to turn large networks of geographically distributed computers into a unified distributed-memory system. In general, such a system will be *heterogeneous*, that is, the individual nodes may be built from different types of hardware.

2.3.3 Interconnection networks

The interconnect plays a decisive role in the performance of both distributed- and shared-memory systems: even if the processors and memory have virtually unlimited performance, a slow interconnect will seriously degrade the overall performance of all but the simplest parallel program. See, for example, Exercise 2.10.

Although some of the interconnects have a great deal in common, there are enough differences to make it worthwhile to treat interconnects for shared-memory and distributed-memory separately.

Shared-memory interconnects

Currently the two most widely used interconnects on shared-memory systems are buses and crossbars. Recall that a **bus** is a collection of parallel communication wires together with some hardware that controls access to the bus. The key characteristic of a bus is that the communication wires are shared by the devices that are connected to it. Buses have the virtue of low cost and flexibility; multiple devices can be connected to a bus with little additional cost. However, since the communication wires are shared, as the number of devices connected to the bus increases, the likelihood that there will be contention for use of the bus increases, and the expected performance of the bus decreases. Therefore, if we connect a large number of processors to a bus, we would expect that the processors would frequently have to wait for access to main memory. Thus, as the size of shared-memory systems increases, buses are rapidly being replaced by *switched* interconnects.

As the name suggests, **switched** interconnects use switches to control the routing of data among the connected devices. A **crossbar** is illustrated in Figure 2.7(a). The lines are bidirectional communication links, the squares are cores or memory modules, and the circles are switches.

The individual switches can assume one of the two configurations shown in Figure 2.7(b). With these switches and at least as many memory modules as processors, there will only be a conflict between two cores attempting to access memory

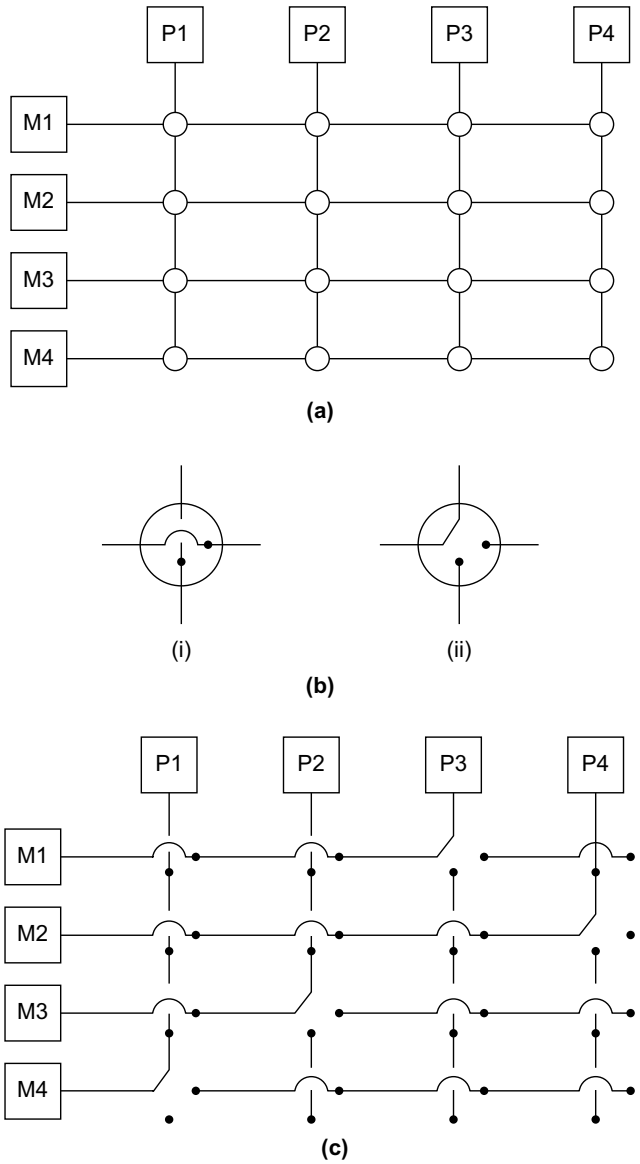


FIGURE 2.7

(a) A crossbar switch connecting four processors (P_i) and four memory modules (M_j); (b) configuration of internal switches in a crossbar; (c) simultaneous memory accesses by the processors

if the two cores attempt to simultaneously access the same memory module. For example, Figure 2.7(c) shows the configuration of the switches if P_1 writes to M_4, P_2 reads from M_3, P_3 reads from M_1, and P_4 writes to M_2.

Crossbars allow simultaneous communication among different devices, so they are much faster than buses. However, the cost of the switches and links is relatively high. A small bus-based system will be much less expensive than a crossbar-based system of the same size.

Distributed-memory interconnects

Distributed-memory interconnects are often divided into two groups: direct interconnects and indirect interconnects. In a **direct interconnect** each switch is directly connected to a processor-memory pair, and the switches are connected to each other. Figure 2.8 shows a **ring** and a two-dimensional **toroidal mesh**. As before, the circles are switches, the squares are processors, and the lines are bidirectional links. A ring is superior to a simple bus since it allows multiple simultaneous communications. However, it's easy to devise communication schemes in which some of the processors must wait for other processors to complete their communications. The toroidal mesh will be more expensive than the ring, because the switches are more complex—they must support five links instead of three—and if there are p processors, the number of links is $3p$ in a toroidal mesh, while it's only $2p$ in a ring. However, it's not difficult to convince yourself that the number of possible simultaneous communications patterns is greater with a mesh than with a ring.

One measure of "number of simultaneous communications" or "connectivity" is **bisection width**. To understand this measure, imagine that the parallel system is

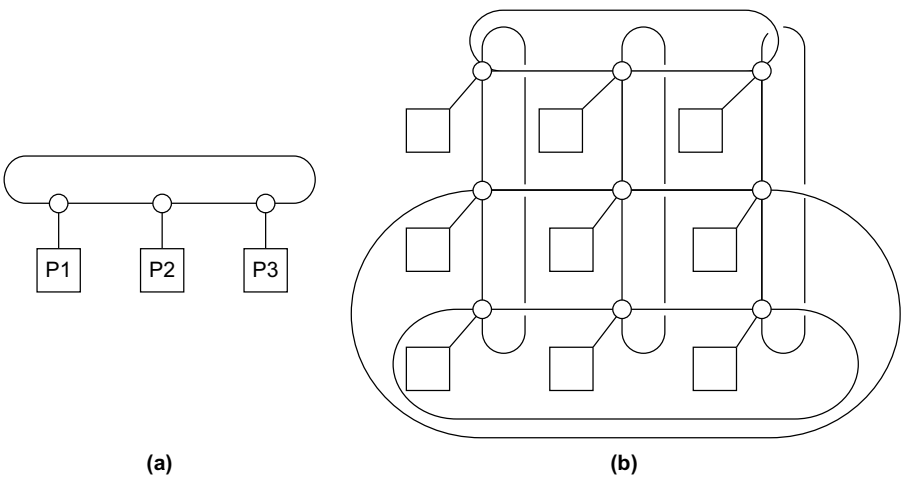


FIGURE 2.8

(a) A ring and (b) a toroidal mesh

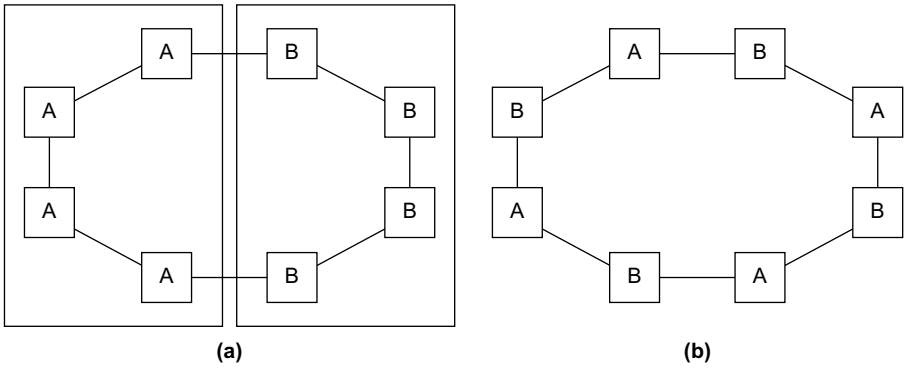


FIGURE 2.9

Two bisections of a ring: (a) only two communications can take place between the halves and (b) four simultaneous connections can take place

divided into two halves, and each half contains half of the processors or nodes. How many simultaneous communications can take place "across the divide" between the halves? In Figure 2.9(a) we've divided a ring with eight nodes into two groups of four nodes, and we can see that only two communications can take place between the halves. (To make the diagrams easier to read, we've grouped each node with its switch in this and subsequent diagrams of direct interconnects.) However, in Figure 2.9(b) we've divided the nodes into two parts so that four simultaneous communications can take place, so what's the bisection width? The bisection width is supposed to give a "worst-case" estimate, so the bisection width is two—not four.

An alternative way of computing the bisection width is to remove the minimum number of links needed to split the set of nodes into two equal halves. The number of links removed is the bisection width. If we have a square two-dimensional toroidal mesh with $p = q^2$ nodes (where q is even), then we can split the nodes into two halves by removing the "middle" horizontal links and the "wraparound" horizontal links. See Figure 2.10. This suggests that the bisection width is at most $2q = 2\sqrt{p}$. In fact, this is the smallest possible number of links and the bisection width of a square two-dimensional toroidal mesh is $2\sqrt{p}$.

The **bandwidth** of a link is the rate at which it can transmit data. It's usually given in megabits or megabytes per second. **Bisection bandwidth** is often used as a measure of network quality. It's similar to bisection width. However, instead of counting the number of links joining the halves, it sums the bandwidth of the links. For example, if the links in a ring have a bandwidth of one billion bits per second, then the bisection bandwidth of the ring will be two billion bits per second or 2000 megabits per second.

The ideal direct interconnect is a **fully connected network** in which each switch is directly connected to every other switch. See Figure 2.11. Its bisection width is $p^2/4$. However, it's impractical to construct such an interconnect for systems with more than a few nodes, since it requires a total of $p^2/2 + p/2$ links, and each switch

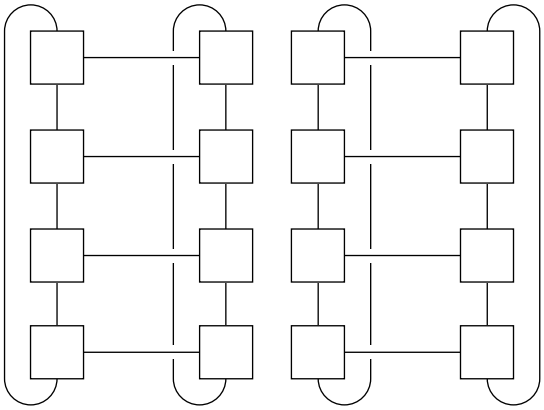


FIGURE 2.10

A bisection of a toroidal mesh

must be capable of connecting to p links. It is therefore more a "theoretical best possible" interconnect than a practical one, and it is used as a basis for evaluating other interconnects.

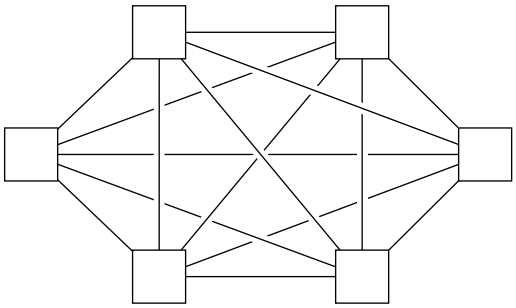


FIGURE 2.11

A fully connected network

The **hypercube** is a highly connected direct interconnect that has been used in actual systems. Hypercubes are built inductively: A one-dimensional hypercube is a fully-connected system with two processors. A two-dimensional hypercube is built from two one-dimensional hypercubes by joining "corresponding" switches. Similarly, a three-dimensional hypercube is built from two two-dimensional hypercubes. See Figure 2.12. Thus, a hypercube of dimension d has $p = 2^d$ nodes, and a switch in a d-dimensional hypercube is directly connected to a processor and d switches. The bisection width of a hypercube is $p/2$, so it has more connectivity than a ring or toroidal mesh, but the switches must be more powerful, since they must support $1 + d = 1 + \log_2(p)$ wires, while the mesh switches only require five wires. So a hypercube with p nodes is more expensive to construct than a toroidal mesh.

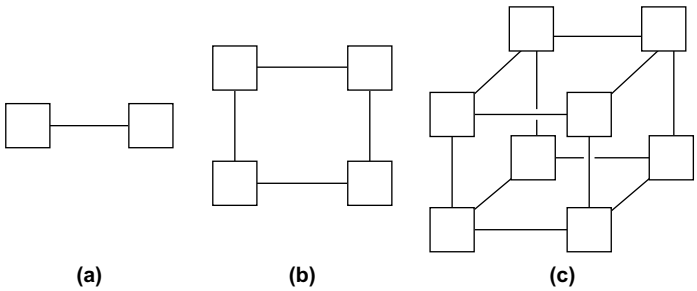


FIGURE 2.12

(a) One-, (b) two-, and (c) three-dimensional hypercubes

Indirect interconnects provide an alternative to direct interconnects. In an indirect interconnect, the switches may not be directly connected to a processor. They're often shown with unidirectional links and a collection of processors, each of which has an outgoing and an incoming link, and a switching network. See Figure 2.13.

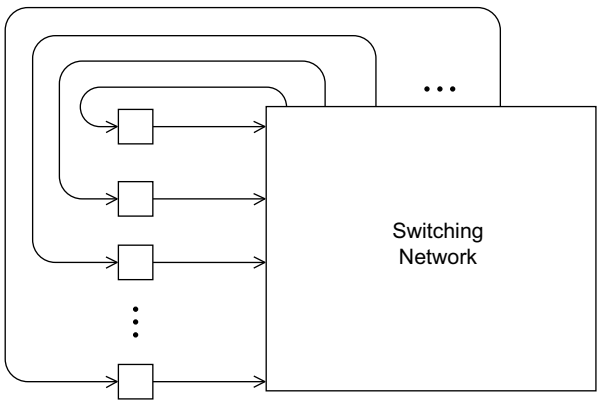


FIGURE 2.13

A generic indirect network

The **crossbar** and the **omega network** are relatively simple examples of indirect networks. We saw a shared-memory crossbar with bidirectional links earlier (Figure 2.7). The diagram of a distributed-memory crossbar in Figure 2.14 has unidirectional links. Notice that as long as two processors don't attempt to communicate with the same processor, all the processors can simultaneously communicate with another processor.

An omega network is shown in Figure 2.15. The switches are two-by-two crossbars (see Figure 2.16). Observe that unlike the crossbar, there are communications that cannot occur simultaneously. For example, in Figure 2.15 if processor 0 sends

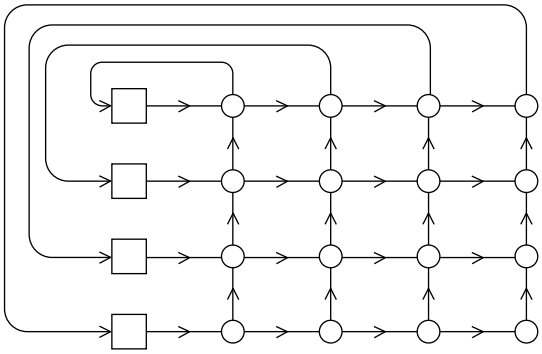


FIGURE 2.14

A crossbar interconnect for distributed-memory

a message to processor 6, then processor 1 cannot simultaneously send a message to processor 7. On the other hand, the omega network is less expensive than the crossbar. The omega network uses $\frac{1}{2}p\log_2(p)$ of the 2×2 crossbar switches, so it uses a total of $2p\log_2(p)$ switches, while the crossbar uses p^2.

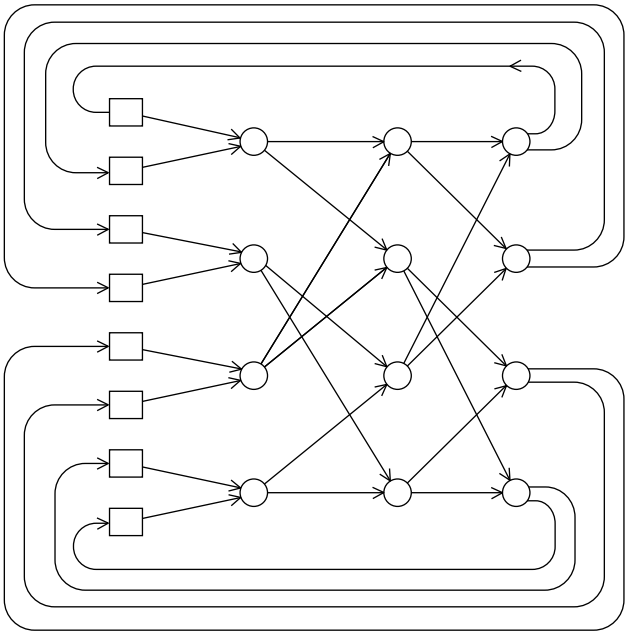


FIGURE 2.15

An omega network

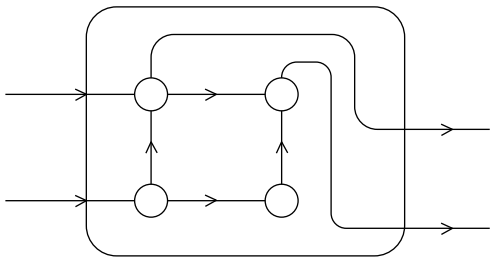


FIGURE 2.16

A switch in an omega network

It's a little bit more complicated to define bisection width for indirect networks. See Exercise 2.14. However, the principle is the same: we want to divide the nodes into two groups of equal size and determine how much communication can take place between the two halves, or alternatively, the minimum number of links that need to be removed so that the two groups can't communicate. The bisection width of a $p \times p$ crossbar is p and the bisection width of an omega network is $p/2$.

Latency and bandwidth

Any time data is transmitted, we're interested in how long it will take for the data to reach its destination. This is true whether we're talking about transmitting data between main memory and cache, cache and register, hard disk and memory, or between two nodes in a distributed-memory or hybrid system. There are two figures that are often used to describe the performance of an interconnect (regardless of what it's connecting): the **latency** and the **bandwidth**. The latency is the time that elapses between the source's beginning to transmit the data and the destination's starting to receive the first byte. The bandwidth is the rate at which the destination receives data after it has started to receive the first byte. So if the latency of an interconnect is l seconds and the bandwidth is b bytes per second, then the time it takes to transmit a message of n bytes is

$$\text{message transmission time} = l + n/b.$$

Beware, however, that these terms are often used in different ways. For example, latency is sometimes used to describe total message transmission time. It's also often used to describe the time required for any fixed overhead involved in transmitting data. For example, if we're sending a message between two nodes in a distributed-memory system, a message is not just raw data. It might include the data to be transmitted, a destination address, some information specifying the size of the message, some information for error correction, and so on. So in this setting, latency might be the time it takes to assemble the message on the sending side—the time needed to combine the various parts—and the time to disassemble the message on the receiving side—the time needed to extract the raw data from the message and store it in its destination.

2.3.4 Cache coherence

Recall that CPU caches are managed by system hardware: programmers don't have direct control over them. This has several important consequences for shared-memory systems. To understand these issues, suppose we have a shared-memory system with two cores, each of which has its own private data cache. See Figure 2.17. As long as the two cores only read shared data, there is no problem. For example, suppose that `x` is a shared variable that has been initialized to 2, `y0` is private and owned by core 0, and `y1` and `z1` are private and owned by core 1. Now suppose the following statements are executed at the indicated times:

Time	Core 0	Core 1
0	`y0 = x;`	`y1 = 3*x;`
1	`x = 7;`	Statement(s) not involving `x`
2	Statement(s) not involving `x`	`z1 = 4*x;`

Then the memory location for `y0` will eventually get the value 2, and the memory location for `y1` will eventually get the value 6. However, it's not so clear what value `z1` will get. It might at first appear that since core 0 updates `x` to 7 before the assignment to `z1`, `z1` will get the value $4 \times 7 = 28$. However, at time 0, `x` is in the cache of core 1. So unless for some reason `x` is evicted from core 0's cache and then reloaded into core 1's cache, it actually appears that the original value `x = 2` may be used, and `z1` will get the value $4 \times 2 = 8$.

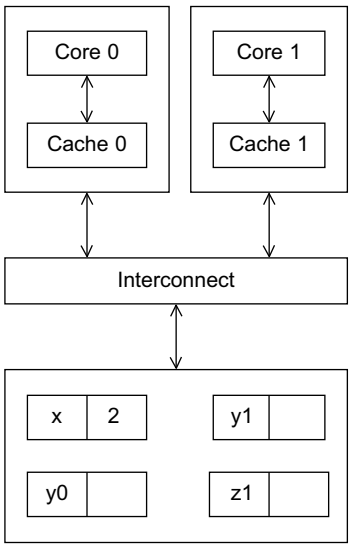


FIGURE 2.17

A shared-memory system with two cores and two caches

Note that this unpredictable behavior will occur regardless of whether the system is using a write-through or a write-back policy. If it's using a write-through policy, the main memory will be updated by the assignment `x = 7`. However, this will have no effect on the value in the cache of core 1. If the system is using a write-back policy, the new value of `x` in the cache of core 0 probably won't even be available to core 1 when it updates `z1`.

Clearly, this is a problem. The programmer doesn't have direct control over when the caches are updated, so her program cannot execute these apparently innocuous statements and know what will be stored in `z1`. There are several problems here, but the one we want to look at right now is that the caches we described for single processor systems provide no mechanism for insuring that when the caches of multiple processors store the same variable, an update by one processor to the cached variable is "seen" by the other processors. That is, that the cached value stored by the other processors is also updated. This is called the **cache coherence** problem.

Snooping cache coherence

There are two main approaches to insuring cache coherence: **snooping cache coherence** and directory-based cache coherence. The idea behind snooping comes from bus-based systems: When the cores share a bus, any signal transmitted on the bus can be "seen" by all the cores connected to the bus. Thus, when core 0 updates the copy of `x` stored in its cache, if it also broadcasts this information across the bus, and if core 1 is "snooping" the bus, it will see that `x` has been updated and it can mark its copy of `x` as invalid. This is more or less how snooping cache coherence works. The principal difference between our description and the actual snooping protocol is that the broadcast only informs the other cores that the *cache line* containing `x` has been updated, not that `x` has been updated.

A couple of points should be made regarding snooping. First, it's not essential that the interconnect be a bus, only that it support broadcasts from each processor to all the other processors. Second, snooping works with both write-through and write-back caches. In principle, if the interconnect is shared—as with a bus—with write-through caches there's no need for additional traffic on the interconnect, since each core can simply "watch" for writes. With write-back caches, on the other hand, an extra communication *is* necessary, since updates to the cache don't get immediately sent to memory.

Directory-based cache coherence

Unfortunately, in large networks broadcasts are expensive, and snooping cache coherence requires a broadcast every time a variable is updated (but see Exercise 2.15). So snooping cache coherence isn't scalable, because for larger systems it will cause performance to degrade. For example, suppose we have a system with the basic distributed-memory architecture (Figure 2.4). However, the system provides a single address space for all the memories. So, for example, core 0 can access the variable `x` stored in core 1's memory, by simply executing a statement such as `y = x`.

(Of course, accessing the memory attached to another core will be slower than accessing "local" memory, but that's another story.) Such a system can, in principle, scale to very large numbers of cores. However, snooping cache coherence is clearly a problem since a broadcast across the interconnect will be very slow relative to the speed of accessing local memory.

Directory-based cache coherence protocols attempt to solve this problem through the use of a data structure called a **directory**. The directory stores the status of each cache line. Typically, this data structure is distributed; in our example, each core/memory pair might be responsible for storing the part of the structure that specifies the status of the cache lines in its local memory. Thus, when a line is read into, say, core 0's cache, the directory entry corresponding to that line would be updated indicating that core 0 has a copy of the line. When a variable is updated, the directory is consulted, and the cache controllers of the cores that have that variable's cache line in their caches are invalidated.

Clearly there will be substantial additional storage required for the directory, but when a cache variable is updated, only the cores storing that variable need to be contacted.

False sharing

It's important to remember that CPU caches are implemented in hardware, so they operate on cache lines, not individual variables. This can have disastrous consequences for performance. As an example, suppose we want to repeatedly call a function `f(i,j)` and add the computed values into a vector:

```
int i, j, m, n;
double y[m];

/* Assign y = 0 */
. . .

for (i = 0; i < m; i++)
   for (j = 0; j < n; j++)
      y[i] += f(i,j);
```

We can parallelize this by dividing the iterations in the outer loop among the cores. If we have `core_count` cores, we might assign the first `m/core_count` iterations to the first core, the next `m/core_count` iterations to the second core, and so on.

```
/* Private variables */
int i, j, iter_count;

/* Shared variables initialized by one core */
int m, n, core_count
double y[m];

iter_count = m/core_count

/* Core 0 does this */
```

```
for (i = 0; i < iter_count; i++)
   for (j = 0; j < n; j++)
      y[i] += f(i,j);

/* Core 1 does this */
for (i = iter_count+1; i < 2*iter_count; i++)
   for (j = 0; j < n; j++)
      y[i] += f(i,j);

. . .
```

Now suppose our shared-memory system has two cores, `m = 8`, doubles are eight bytes, cache lines are 64 bytes, and `y[0]` is stored at the beginning of a cache line. A cache line can store eight doubles, and `y` takes one full cache line. What happens when core 0 and core 1 simultaneously execute their codes? Since all of `y` is stored in a single cache line, each time one of the cores executes the statement `y[i] += f(i,j)`, the line will be invalidated, and the next time the other core tries to execute this statement it will have to fetch the updated line from memory! So if n is large, we would expect that a large percentage of the assignments `y[i] += f(i,j)` will access main memory—in spite of the fact that core 0 and core 1 never access each others' elements of `y`. This is called **false sharing**, because the system is behaving *as if* the elements of `y` were being shared by the cores.

Note that false sharing does not cause incorrect results. However, it can ruin the performance of a program by causing many more accesses to memory than necessary. We can reduce its effect by using temporary storage that is local to the thread or process and then copying the temporary storage to the shared storage. We'll return to the subject of false sharing in Chapters 4 and 5.

2.3.5 Shared-memory versus distributed-memory

Newcomers to parallel computing sometimes wonder why all MIMD systems aren't shared-memory, since most programmers find the concept of implicitly coordinating the work of the processors through shared data structures more appealing than explicitly sending messages. There are several issues, some of which we'll discuss when we talk about software for distributed- and shared-memory. However, the principal hardware issue is the cost of scaling the interconnect. As we add processors to a bus, the chance that there will be conflicts over access to the bus increase dramatically, so buses are suitable for systems with only a few processors. Large crossbars are very expensive, so it's also unusual to find systems with large crossbar interconnects. On the other hand, distributed-memory interconnects such as the hypercube and the toroidal mesh are relatively inexpensive, and distributed-memory systems with thousands of processors that use these and other interconnects have been built. Thus, distributed-memory systems are often better suited for problems requiring vast amounts of data or computation.

2.4 PARALLEL SOFTWARE

Parallel hardware has arrived. Virtually all desktop and server systems use multicore processors. The same cannot be said for parallel software. Except for operating systems, database systems, and Web servers, there is currently very little commodity software that makes extensive use of parallel hardware. As we noted in Chapter 1, this is a problem because we can no longer rely on hardware and compilers to provide a steady increase in application performance. If we're to continue to have routine increases in application performance and application power, software developers must learn to write applications that exploit shared- and distributed-memory architectures. In this section we'll take a look at some of the issues involved in writing software for parallel systems.

First, some terminology. Typically when we run our shared-memory programs, we'll start a single process and fork multiple threads. So when we discuss shared-memory programs, we'll talk about *threads* carrying out tasks. On the other hand, when we run distributed-memory programs, we'll start multiple processes, and we'll talk about *processes* carrying out tasks. When the discussion applies equally well to shared-memory and distributed-memory systems, we'll talk about *processes/threads* carrying out tasks.

2.4.1 Caveats

Before proceeding, we need to stress some of the limitations of this section. First, here, and in the remainder of the book, we'll only be discussing software for MIMD systems. For example, while the use of GPUs as a platform for parallel computing continues to grow at a rapid pace, the application programming interfaces (APIs) for GPUs are necessarily very different from standard MIMD APIs. Second, we stress that our coverage is only meant to give some idea of the issues: there is no attempt to be comprehensive.

Finally, we'll mainly focus on what's often called **single program, multiple data**, or SPMD, programs. Instead of running a different program on each core, SPMD programs consist of a single executable that can behave as if it were multiple different programs through the use of conditional branches. For example,

```
if (I'm thread/process 0)
   do this;
else
   do that;
```

Observe that SPMD programs can readily implement data-parallelism. For example,

```
if (I'm thread/process 0)
   operate on the first half of the array;
else /* I'm thread/process 1 */
   operate on the second half of the array;
```

Recall that a program is **task parallel** if it obtains its parallelism by dividing tasks among the threads or processes. The first example makes it clear that SPMD programs can also implement **task-parallelism**.

2.4.2 Coordinating the processes/threads

In a very few cases, obtaining excellent parallel performance is trivial. For example, suppose we have two arrays and we want to add them:

```
double x[n], y[n];
. . .
for (int i = 0; i < n; i++)
   x[i] += y[i];
```

In order to parallelize this, we only need to assign elements of the arrays to the processes/threads. For example, if we have p processes/threads, we might make process/thread 0 responsible for elements $0,\ldots,n/p-1$, process/thread 1 would be responsible for elements $n/p,\ldots,2n/p-1$, and so on.

So for this example, the programmer only needs to

1. Divide the work among the processes/threads
   a. in such a way that each process/thread gets roughly the same amount of work, and
   b. in such a way that the amount of communication required is minimized.

   Recall that the process of dividing the work among the processes/threads so that (a) is satisfied is called **load balancing**. The two qualifications on dividing the work are obvious, but nonetheless important. In many cases it won't be necessary to give much thought to them; they typically become concerns in situations in which the amount of work isn't known in advance by the programmer, but rather the work is generated as the program runs. For an example, see the tree search problem in Chapter 6.

   Although we might wish for a term that's a little easier to pronounce, recall that the process of converting a serial program or algorithm into a parallel program is often called **parallelization**. Programs that can be parallelized by simply dividing the work among the processes/threads are sometimes said to be **embarrassingly parallel**. This is a bit unfortunate, since it suggests that programmers should be embarrassed to have written an embarrassingly parallel program, when, to the contrary, successfully devising a parallel solution to *any* problem is a cause for great rejoicing.

   Alas, the vast majority of problems are much more determined to resist our efforts to find a parallel solution. As we saw in Chapter 1, for these problems, we need to coordinate the work of the processes/threads. In these programs, we also usually need to

2. Arrange for the processes/threads to synchronize.
3. Arrange for communication among the processes/threads.

These last two problems are often interrelated. For example, in distributed-memory programs, we often implicitly synchronize the processes by communicating among

them, and in shared-memory programs, we often communicate among the threads by synchronizing them. We'll say more about both issues below.

2.4.3 Shared-memory

As we noted earlier, in shared-memory programs, variables can be **shared** or **private**. Shared variables can be read or written by any thread, and private variables can ordinarily only be accessed by one thread. Communication among the threads is usually done through shared variables, so communication is implicit, rather than explicit.

Dynamic and static threads

In many environments shared-memory programs use **dynamic threads**. In this paradigm, there is often a master thread and at any given instant a (possibly empty) collection of worker threads. The master thread typically waits for work requests—for example, over a network—and when a new request arrives, it forks a worker thread, the thread carries out the request, and when the thread completes the work, it terminates and joins the master thread. This paradigm makes efficient use of system resources since the resources required by a thread are only being used while the thread is actually running.

An alternative to the dynamic paradigm is the **static thread** paradigm. In this paradigm, all of the threads are forked after any needed setup by the master thread and the threads run until all the work is completed. After the threads join the master thread, the master thread may do some cleanup (e.g., free memory) and then it also terminates. In terms of resource usage, this may be less efficient: if a thread is idle, its resources (e.g., stack, program counter, and so on.) can't be freed. However, forking and joining threads can be fairly time-consuming operations. So if the necessary resources are available, the static thread paradigm has the potential for better performance than the dynamic paradigm. It also has the virtue that it's closer to the most widely used paradigm for distributed-memory programming, so part of the mindset that is used for one type of system is preserved for the other. Hence, we'll often use the static thread paradigm.

Nondeterminism

In any MIMD system in which the processors execute asynchronously it is likely that there will be **nondeterminism**. A computation is nondeterministic if a given input can result in different outputs. If multiple threads are executing independently, the relative rate at which they'll complete statements varies from run to run, and hence the results of the program may be different from run to run. As a very simple example, suppose we have two threads, one with id or rank 0 and the other with id or rank 1. Suppose also that each is storing a private variable `my_x`, thread 0's value for `my_x` is 7, and thread 1's is 19. Further, suppose both threads execute the following code:

```
. . .
printf("Thread %d > my_val = %d\n", my_rank, my_x);
. . .
```

Then the output could be

```
Thread 0 > my_val = 7
Thread 1 > my_val = 19
```

but it could also be

```
Thread 1 > my_val = 19
Thread 0 > my_val = 7
```

In fact, things could be even worse: the output of one thread could be broken up by the output of the other thread. However, the point here is that because the threads are executing independently and interacting with the operating system, the time it takes for one thread to complete a block of statements varies from execution to execution, so the order in which these statements complete can't be predicted.

In many cases nondeterminism isn't a problem. In our example, since we've labelled the output with the thread's rank, the order in which the output appears probably doesn't matter. However, there are also many cases in which nondeterminism—especially in shared-memory programs—can be disastrous, because it can easily result in program errors. Here's a simple example with two threads.

Suppose each thread computes an `int`, which it stores in a private variable `my_val`. Suppose also that we want to add the values stored in `my_val` into a shared-memory location `x` that has been initialized to 0. Both threads therefore want to execute code that looks something like this:

```
my_val = Compute_val(my_rank);
x += my_val;
```

Now recall that an addition typically requires loading the two values to be added into registers, adding the values, and finally storing the result. To keep things relatively simple, we'll assume that values are loaded from main memory directly into registers and stored in main memory directly from registers. Here is one possible sequence of events:

Time	Core 0	Core 1
0	Finish assignment to `my_val`	In call to `Compute_val`
1	Load `x = 0` into register	Finish assignment to `my_val`
2	Load `my_val = 7` into register	Load `x = 0` into register
3	Add `my_val = 7` to `x`	Load `my_val = 19` into register
4	Store `x = 7`	Add `my_val` to `x`
5	Start other work	Store `x = 19`

Clearly this is not what we want, and it's easy to imagine other sequences of events that result in an incorrect value for `x`. The nondeterminism here is a result of the fact that two threads are attempting to more or less simultaneously update the memory location `x`. When threads or processes attempt to simultaneously access a

resource, and the accesses can result in an error, we often say the program has a **race condition**, because the threads or processes are in a "horse race." That is, the outcome of the computation depends on which thread wins the race. In our example, the threads are in a race to execute `x += my_val`. In this case, unless one thread completes `x += my_val` before the other thread starts, the result will be incorrect. A block of code that can only be executed by one thread at a time is called a **critical section**, and it's usually our job as programmers to insure **mutually exclusive** access to the critical section. In other words, we need to insure that if one thread is executing the code in the critical section, then the other threads are excluded.

The most commonly used mechanism for insuring mutual exclusion is a **mutual exclusion lock** or **mutex** or **lock**. A mutex is a special type of object that has support in the underlying hardware. The basic idea is that each critical section is *protected* by a lock. Before a thread can execute the code in the critical section, it must "obtain" the mutex by calling a mutex function, and, when it's done executing the code in the critical section, it should "relinquish" the mutex by calling an unlock function. While one thread "owns" the lock—that is, has returned from a call to the lock function, but hasn't yet called the unlock function—any other thread attempting to execute the code in the critical section will wait in its call to the lock function.

Thus, in order to insure that our code functions correctly, we might modify it so that it looks something like this:

```
my_val = Compute_val(my_rank);
Lock(&add_my_val_lock);
x += my_val;
Unlock(&add_my_val_lock);
```

This insures that only one thread at a time can execute the statement `x += my_val`. Note that the code does *not* impose any predetermined order on the threads. Either thread 0 or thread 1 can execute `x += my_val` first.

Also note that the use of a mutex enforces **serialization** of the critical section. Since only one thread at a time can execute the code in the critical section, this code is effectively serial. Thus, we want our code to have as few critical sections as possible, and we want our critical sections to be as short as possible.

There are alternatives to mutexes. In **busy-waiting**, a thread enters a loop whose sole purpose is to test a condition. In our example, suppose there is a shared variable `ok_for_1` that has been initialized to false. Then something like the following code can insure that thread 1 won't update `x` until after thread 0 has updated it:

```
my_val = Compute_val(my_rank);
if (my_rank == 1)
   while (!ok_for_1);  /* Busy-wait loop */
x += my_val;           /* Critical section */
if (my_rank == 0)
   ok_for_1 = true;    /* Let thread 1 update x */
```

So until thread 0 executes `ok_for_1 = true`, thread 1 will be stuck in the loop `while (!ok_for_1)`. This loop is called a "busy-wait" because the thread can be

very busy waiting for the condition. This has the virtue that it's simple to understand and implement. However, it can be very wasteful of system resources, because even when a thread is doing no useful work, the core running the thread will be repeatedly checking to see if the critical section can be entered. **Semaphores** are similar to mutexes, although the details of their behavior are slightly different, and there are some types of thread synchronization that are easier to implement with semaphores than mutexes. A **monitor** provides mutual exclusion at a somewhat higher-level: it is an object whose methods can only be executed by one thread at a time. We'll discuss busy-waiting and semaphores in Chapter 4.

There are a number of other alternatives that are currently being studied but that are not yet widely available. The one that has attracted the most attention is probably **transactional memory** [31]. In database management systems, a **transaction** is an access to a database that the system treats as a single unit. For example, transferring $1000 from your savings account to your checking account should be treated by your bank's software as a transaction, so that the software can't debit your savings account without also crediting your checking account. If the software was able to debit your savings account, but was then unable to credit your checking account, it would *roll-back* the transaction. In other words, the transaction would either be fully completed or any partial changes would be erased. The basic idea behind transactional memory is that critical sections in shared-memory programs should be treated as transactions. Either a thread successfully completes the critical section or any partial results are rolled back and the critical section is repeated.

Thread safety

In many, if not most, cases parallel programs can call functions developed for use in serial programs, and there won't be any problems. However, there are some notable exceptions. The most important exception for C programmers occurs in functions that make use of *static* local variables. Recall that ordinary C local variables—variables declared inside a function—are allocated from the system stack. Since each thread has its own stack, ordinary C local variables are private. However, recall that a static variable that's declared in a function persists from one call to the next. Thus, static variables are effectively shared among any threads that call the function, and this can have unexpected and unwanted consequences.

For example, the C string library function `strtok` splits an input string into substrings. When it's first called, it's passed a string, and on subsequent calls it returns successive substrings. This can be arranged through the use of a static `char*` variable that refers to the string that was passed on the first call. Now suppose two threads are splitting strings into substrings. Clearly, if, for example, thread 0 makes its first call to `strtok`, and then thread 1 makes its first call to `strtok` before thread 0 has completed splitting its string, then thread 0's string will be lost or overwritten, and, on subsequent calls it may get substrings of thread 1's strings.

A function such as `strtok` is not **thread safe**. This means that if it is used in a multithreaded program, there may be errors or unexpected results. When a block of code isn't thread safe, it's usually because different threads are accessing shared

data. Thus, as we've seen, even though many serial functions can be used safely in multithreaded programs—that is, they're *thread safe*—programmers need to be wary of functions that were written exclusively for use in serial programs. We'll take a closer look at thread safety in Chapters 4 and 5.

2.4.4 Distributed-memory

In distributed-memory programs, the cores can directly access only their own, private memories. There are several APIs that are used. However, by far the most widely used is message-passing. So we'll devote most of our attention in this section to message-passing. Then we'll take a brief look at at a couple of other, less widely used, APIs.

Perhaps the first thing to note regarding distributed-memory APIs is that they can be used with shared-memory hardware. It's perfectly feasible for programmers to logically partition shared-memory into private address spaces for the various threads, and a library or compiler can implement the communication that's needed.

As we noted earlier, distributed-memory programs are usually executed by starting multiple processes rather than multiple threads. This is because typical "threads of execution" in a distributed-memory program may run on independent CPUs with independent operating systems, and there may be no software infrastructure for starting a single "distributed" process and having that process fork one or more threads on each node of the system.

Message-passing

A message-passing API provides (at a minimum) a send and a receive function. Processes typically identify each other by ranks in the range $0, 1, \ldots, p-1$, where p is the number of processes. So, for example, process 1 might send a message to process 0 with the following pseudo-code:

```
char message[100];
. . .
my_rank = Get_rank();
if (my_rank == 1) {
   sprintf(message, "Greetings from process 1");
   Send(message, MSG_CHAR, 100, 0);
} else if (my_rank == 0) {
   Receive(message, MSG_CHAR, 100, 1);
   printf("Process 0 > Received: %s\n", message);
}
```

Here the `Get_rank` function returns the calling process' rank. Then the processes branch depending on their ranks. Process 1 creates a message with `sprintf` from the standard C library and then sends it to process 0 with the call to `Send`. The arguments to the call are, in order, the message, the type of the elements in the message (`MSG_CHAR`), the number of elements in the message (100), and the rank of the destination process (0). On the other hand, process 0 calls `Receive` with the following arguments: the variable into which the message will be received (`message`), the

type of the message elements, the number of elements available for storing the message, and the rank of the process sending the message. After completing the call to `Receive`, process 0 prints the message.

Several points are worth noting here. First note that the program segment is SPMD. The two processes are using the same executable, but carrying out different actions. In this case, what they do depends on their ranks. Second, note that the variable `message` refers to different blocks of memory on the different processes. Programmers often stress this by using variable names such as `my_message` or `local_message`. Finally, note that we're assuming that process 0 can write to `stdout`. This is usually the case: most implementations of message-passing APIs allow all processes access to `stdout` and `stderr`—even if the API doesn't explicitly provide for this. We'll talk a little more about I/O later on.

There are several possibilities for the exact behavior of the `Send` and `Receive` functions, and most message-passing APIs provide several different send and/or receive functions. The simplest behavior is for the call to `Send` to **block** until the call to `Receive` starts receiving the data. This means that the process calling `Send` won't return from the call until the matching call to `Receive` has started. Alternatively, the `Send` function may copy the contents of the message into storage that it owns, and then it will return as soon as the data is copied. The most common behavior for the `Receive` function is for the receiving process to block until the message is received. There are other possibilities for both `Send` and `Receive`, and we'll discuss some of them in Chapter 3.

Typical message-passing APIs also provide a wide variety of additional functions. For example, there may be functions for various "collective" communications, such as a **broadcast**, in which a single process transmits the same data to all the processes, or a **reduction**, in which results computed by the individual processes are combined into a single result—for example, values computed by the processes are added. There may also be special functions for managing processes and communicating complicated data structures. The most widely used API for message-passing is the **Message-Passing Interface** or MPI. We'll take a closer look at it in Chapter 3.

Message-passing is a very powerful and versatile API for developing parallel programs. Virtually all of the programs that are run on the most powerful computers in the world use message-passing. However, it is also very low level. That is, there is a huge amount of detail that the programmer needs to manage. For example, in order to parallelize a serial program, it is usually necessary to rewrite the vast majority of the program. The data structures in the program may have to either be replicated by each process or be explicitly distributed among the processes. Furthermore, the rewriting usually can't be done incrementally. For example, if a data structure is used in many parts of the program, distributing it for the parallel parts and collecting it for the serial (unparallelized) parts will probably be prohibitively expensive. Therefore, message-passing is sometimes called "the assembly language of parallel programming," and there have been many attempts to develop other distributed-memory APIs.

One-sided communication

In message-passing, one process, must call a send function and the send must be matched by another process' call to a receive function. Any communication requires the explicit participation of two processes. In **one-sided communication**, or **remote memory access**, a single process calls a function, which updates either local memory with a value from another process or remote memory with a value from the calling process. This can simplify communication, since it only requires the active participation of a single process. Furthermore, it can significantly reduce the cost of communication by eliminating the overhead associated with synchronizing two processes. It can also reduce overhead by eliminating the overhead of one of the function calls (send or receive).

It should be noted that some of these advantages may be hard to realize in practice. For example, if process 0 is copying a value into the memory of process 1, 0 must have some way of knowing when it's safe to copy, since it will overwrite some memory location. Process 1 must also have some way of knowing when the memory location has been updated. The first problem can be solved by synchronizing the two processes before the copy, and the second problem can be solved by another synchronization or by having a "flag" variable that process 0 sets after it has completed the copy. In the latter case, process 1 may need to **poll** the flag variable in order to determine that the new value is available. That is, it must repeatedly check the flag variable until it gets the value indicating 0 has completed its copy. Clearly, these problems can considerably increase the overhead associated with transmitting a value. A further difficulty is that since there is no explicit interaction between the two processes, remote memory operations can introduce errors that are very hard to track down.

Partitioned global address space languages

Since many programmers find shared-memory programming more appealing than message-passing or one-sided communication, a number of groups are developing parallel programming languages that allow the user to use some shared-memory techniques for programming distributed-memory hardware. This isn't quite as simple as it sounds. If we simply wrote a compiler that treated the collective memories in a distributed-memory system as a single large memory, our programs would have poor, or, at best, unpredictable performance, since each time a running process accessed memory, it might access local memory—that is, memory belonging to the core on which it was executing—or remote memory, memory belonging to another core. Accessing remote memory can take hundreds or even thousands of times longer than accessing local memory. As an example, consider the following pseudo-code for a shared-memory vector addition:

```
shared int n = . . . ;
shared double x[n], y[n];
private int i, my_first_element, my_last_element;
my_first_element = . . . ;
my_last_element = . . . ;
```

```
/* Initialize x and y */
. . .

for (i = my_first_element; i <= my_last_element; i++)
   x[i] += y[i];
```

We first declare two shared arrays. Then, on the basis of the process' rank, we determine which elements of the array "belong" to which process. After initializing the arrays, each process adds its assigned elements. If the assigned elements of `x` and `y` have been allocated so that the elements assigned to each process are in the memory attached to the core the process is running on, then this code should be very fast. However, if, for example, all of `x` is assigned to core 0 and all of `y` is assigned to core 1, then the performance is likely to be terrible, since each time the assignment `x[i] += y[i]` is executed, the process will need to refer to remote memory.

Partitioned global address space, or PGAS, languages provide some of the mechanisms of shared-memory programs. However, they provide the programmer with tools to avoid the problem we just discussed. Private variables are allocated in the local memory of the core on which the process is executing, and the distribution of the data in shared data structures is controlled by the programmer. So, for example, she knows which elements of a shared array are in which process' local memory.

There are a number of research projects currently working on the development of PGAS languages. See, for example, [7, 9, 45].

2.4.5 Programming hybrid systems

Before moving on, we should note that it is possible to program systems such as clusters of multicore processors using a combination of a shared-memory API on the nodes and a distributed-memory API for internode communication. However, this is usually only done for programs that require the highest possible levels of performance, since the complexity of this "hybrid" API makes program development extremely difficult. See, for example, [40]. Rather, such systems are usually programmed using a single, distributed-memory API for both inter- and intra-node communication.

2.5 INPUT AND OUTPUT

We've generally avoided the issue of input and output. There are a couple of reasons. First and foremost, parallel I/O, in which multiple cores access multiple disks or other devices, is a subject to which one could easily devote a book. See, for example, [35]. Second, the vast majority of the programs we'll develop do very little in the way of I/O. The amount of data they read and write is quite small and

easily managed by the standard C I/O functions `printf`, `fprintf`, `scanf`, and `fscanf`. However, even the limited use we make of these functions can potentially cause some problems. Since these functions are part of standard C, which is a serial language, the standard says nothing about what happens when they're called by different processes. On the other hand, threads that are forked by a single process *do* share `stdin`, `stdout`, and `stderr`. However, (as we've seen), when multiple threads attempt to access one of these, the outcome is nondeterministic, and it's impossible to predict what will happen.

When we call `printf` from multiple processes, we, as developers, would like the output to appear on the console of a single system, the system on which we started the program. In fact, this is what the vast majority of systems do. However, there is no guarantee, and we need to be aware that it is possible for a system to do something else, for example, only one process has access to `stdout` or `stderr` or even *no* processes have access to `stdout` or `stderr`.

What *should* happen with calls to `scanf` when we're running multiple processes is a little less obvious. Should the input be divided among the processes? Or should only a single process be allowed to call `scanf`? The vast majority of systems allow at least one process to call `scanf`—usually process 0—while some allow more processes. Once again, there are some systems that don't allow any processes to call `scanf`.

When multiple processes *can* access `stdout`, `stderr`, or `stdin`, as you might guess, the distribution of the input and the sequence of the output are usually nondeterministic. For output, the data will probably appear in a different order each time the program is run, or, even worse, the output of one process may be broken up by the output of another process. For input, the data read by each process may be different on each run, even if the same input is used.

In order to partially address these issues, we'll be making these assumptions and following these rules when our parallel programs need to do I/O:

- In distributed-memory programs, only process 0 will access `stdin`. In shared-memory programs, only the master thread or thread 0 will access `stdin`.
- In both distributed-memory and shared-memory programs, all the processes/threads can access `stdout` and `stderr`.
- However, because of the nondeterministic order of output to `stdout`, in most cases only a single process/thread will be used for all output to `stdout`. The principal exception will be output for debugging a program. In this situation, we'll often have multiple processes/threads writing to `stdout`.
- Only a single process/thread will attempt to access any single file other than `stdin`, `stdout`, or `stderr`. So, for example, each process/thread can open its own, private file for reading or writing, but no two processes/threads will open the same file.
- Debug output should always include the rank or id of the process/thread that's generating the output.

2.6 PERFORMANCE

Of course our main purpose in writing parallel programs is usually increased performance. So what can we expect? And how can we evaluate our programs?

2.6.1 Speedup and efficiency

Usually the best we can hope to do is to equally divide the work among the cores, while at the same time introducing no additional work for the cores. If we succeed in doing this, and we run our program with p cores, one thread or process on each core, then our parallel program will run p times faster than the serial program. If we call the serial run-time T_{serial} and our parallel run-time T_{parallel}, then the best we can hope for is $T_{\text{parallel}} = T_{\text{serial}}/p$. When this happens, we say that our parallel program has **linear speedup**.

In practice, we're unlikely to get linear speedup because the use of multiple processes/threads almost invariably introduces some overhead. For example, shared-memory programs will almost always have critical sections, which will require that we use some mutual exclusion mechanism such as a mutex. The calls to the mutex functions are overhead that's not present in the serial program, and the use of the mutex forces the parallel program to serialize execution of the critical section. Distributed-memory programs will almost always need to transmit data across the network, which is usually much slower than local memory access. Serial programs, on the other hand, won't have these overheads. Thus, it will be very unusual for us to find that our parallel programs get linear speedup. Furthermore, it's likely that the overheads will increase as we increase the number of processes or threads, that is, more threads will probably mean more threads need to access a critical section. More processes will probably mean more data needs to be transmitted across the network.

So if we define the **speedup** of a parallel program to be

$$S = \frac{T_{\text{serial}}}{T_{\text{parallel}}},$$

then linear speedup has $S = p$, which is unusual. Furthermore, as p increases, we expect S to become a smaller and smaller fraction of the ideal, linear speedup p. Another way of saying this is that S/p will probably get smaller and smaller as p increases. Table 2.4 shows an example of the changes in S and S/p as p increases.[1]

This value, S/p, is sometimes called the **efficiency** of the parallel program. If we substitute the formula for S, we see that the efficiency is

$$E = \frac{S}{p} = \frac{\left(\frac{T_{\text{serial}}}{T_{\text{parallel}}}\right)}{p} = \frac{T_{\text{serial}}}{p \cdot T_{\text{parallel}}}.$$

[1] These data are taken from Chapter 3. See Tables 3.6 and 3.7.

Table 2.4 Speedups and Efficiencies of a Parallel Program

p	1	2	4	8	16
S	1.0	1.9	3.6	6.5	10.8
E = S/p	1.0	0.95	0.90	0.81	0.68

Table 2.5 Speedups and Efficiencies of a Parallel Program on Different Problem Sizes

	p	1	2	4	8	16
Half	S	1.0	1.9	3.1	4.8	6.2
	E	1.0	0.95	0.78	0.60	0.39
Original	S	1.0	1.9	3.6	6.5	10.8
	E	1.0	0.95	0.90	0.81	0.68
Double	S	1.0	1.9	3.9	7.5	14.2
	E	1.0	0.95	0.98	0.94	0.89

It's clear that T_{parallel}, S, and E depend on p, the number of processes or threads. We also need to keep in mind that T_{parallel}, S, E, and T_{serial} all depend on the problem size. For example, if we halve and double the problem size of the program whose speedups are shown in Table 2.4, we get the speedups and efficiencies shown in Table 2.5. The speedups are plotted in Figure 2.18, and the efficiencies are plotted in Figure 2.19.

We see that in this example, when we increase the problem size, the speedups and the efficiencies increase, while they decrease when we decrease the problem size. This behavior is quite common. Many parallel programs are developed by dividing the work of the serial program among the processes/threads and adding in the necessary "parallel overhead" such as mutual exclusion or communication. Therefore, if T_{overhead} denotes this parallel overhead, it's often the case that

$$T_{\text{parallel}} = T_{\text{serial}}/p + T_{\text{overhead}}.$$

Furthermore, as the problem size is increased, T_{overhead} often grows more slowly than T_{serial}. When this is the case the speedup and the efficiency will increase. See Exercise 2.16. This is what your intuition should tell you: there's more work for the processes/threads to do, so the relative amount of time spent coordinating the work of the processes/threads should be less.

A final issue to consider is what values of T_{serial} should be used when reporting speedups and efficiencies. Some authors say that T_{serial} should be the run-time of the fastest program on the fastest processor available. In practice, most authors use a serial program on which the parallel program was based and run it on a single

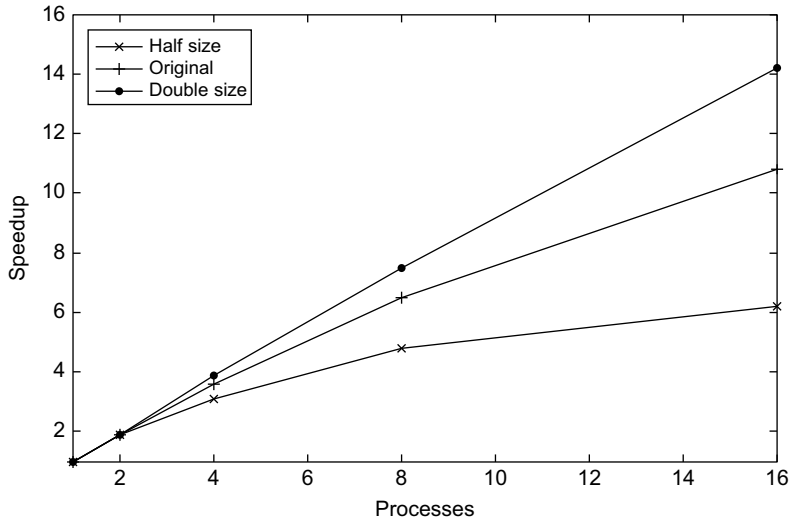


FIGURE 2.18

Speedups of parallel program on different problem sizes

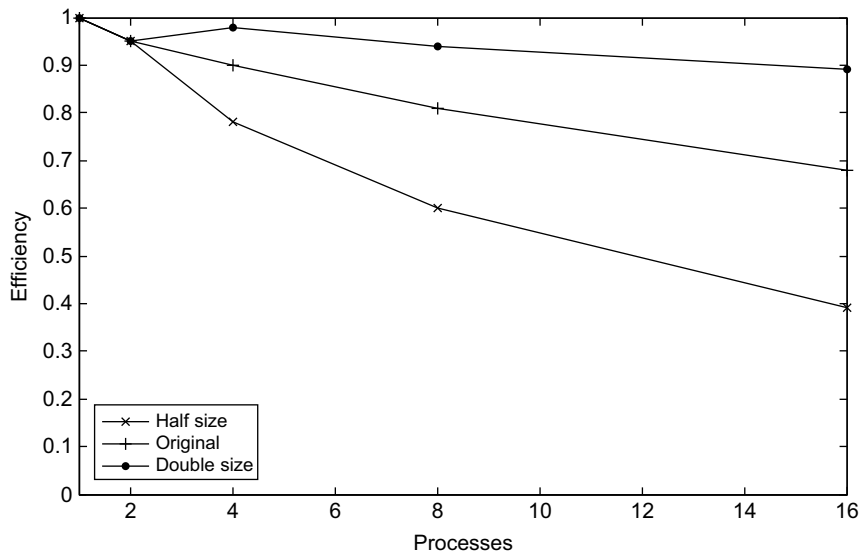


FIGURE 2.19

Efficiencies of parallel program on different problem sizes

processor of the parallel system. So if we were studying the performance of a parallel shell sort program, authors in the first group might use a serial radix sort or quicksort on a single core of the fastest system available, while authors in the second group would use a serial shell sort on a single processor of the parallel system. We'll generally use the second approach.

2.6.2 Amdahl's law

Back in the 1960s, Gene Amdahl made an observation [2] that's become known as **Amdahl's law.** It says, roughly, that unless virtually all of a serial program is parallelized, the possible speedup is going to be very limited—regardless of the number of cores available. Suppose, for example, that we're able to parallelize 90% of a serial program. Further suppose that the parallelization is "perfect," that is, regardless of the number of cores p we use, the speedup of this part of the program will be p. If the serial run-time is T_{serial}= 20 seconds, then the run-time of the parallelized part will be $0.9 \times T_{\text{serial}}/p = 18/p$ and the run-time of the "unparallelized" part will be $0.1 \times T_{\text{serial}} = 2$. The overall parallel run-time will be

$$T_{\text{parallel}} = 0.9 \times T_{\text{serial}}/p + 0.1 \times T_{\text{serial}} = 18/p + 2,$$

and the speedup will be

$$S = \frac{T_{\text{serial}}}{0.9 \times T_{\text{serial}}/p + 0.1 \times T_{\text{serial}}} = \frac{20}{18/p + 2}.$$

Now as p gets larger and larger, $0.9 \times T_{\text{serial}}/p = 18/p$ gets closer and closer to 0, so the total parallel run-time can't be smaller than $0.1 \times T_{\text{serial}} = 2$. That is, the denominator in S can't be smaller than $0.1 \times T_{\text{serial}} = 2$. The fraction S must therefore be smaller than

$$S \leq \frac{T_{\text{serial}}}{0.1 \times T_{\text{serial}}} = \frac{20}{2} = 10.$$

That is, $S \leq 10$. This is saying that even though we've done a perfect job in parallelizing 90% of the program, and even if we have, say, 1000 cores, we'll never get a speedup better than 10.

More generally, if a fraction r of our serial program remains unparallelized, then Amdahl's law says we can't get a speedup better than $1/r$. In our example, $r = 1 - 0.9 = 1/10$, so we couldn't get a speedup better than 10. Therefore, if a fraction r of our serial program is "inherently serial," that is, cannot possibly be parallelized, then we can't possibly get a speedup better than $1/r$. Thus, even if r is quite small—say 1/100—and we have a system with thousands of cores, we can't possibly get a speedup better than 100.

This is pretty daunting. Should we give up and go home? Well, no. There are several reasons not to be too worried by Amdahl's law. First, it doesn't take into consideration the problem size. For many problems, as we increase the problem size,

the "inherently serial" fraction of the program decreases in size; a more mathematical version of this statement is known as **Gustafson's law** [25]. Second, there are thousands of programs used by scientists and engineers that routinely obtain huge speedups on large distributed-memory systems. Finally, is a small speedup so awful? In many cases, obtaining a speedup of 5 or 10 is more than adequate, especially if the effort involved in developing the parallel program wasn't very large.

2.6.3 Scalability

The word "scalable" has a wide variety of informal uses. Indeed, we've used it several times already. Roughly speaking, a technology is scalable if it can handle ever-increasing problem sizes. However, in discussions of parallel program performance, scalability has a somewhat more formal definition. Suppose we run a parallel program with a fixed number of processes/threads and a fixed input size, and we obtain an efficiency E. Suppose we now increase the number of processes/threads that are used by the program. If we can find a corresponding rate of increase in the problem size so that the program always has efficiency E, then the program is **scalable**.

As an example, suppose that $T_{\text{serial}} = n$, where the units of T_{serial} are in microseconds, and n is also the problem size. Also suppose that $T_{\text{parallel}} = n/p + 1$. Then

$$E = \frac{n}{p(n/p+1)} = \frac{n}{n+p}.$$

To see if the program is scalable, we increase the number of processes/threads by a factor of k, and we want to find the factor x that we need to increase the problem size by so that E is unchanged. The number of processes/threads will be kp and the problem size will be xn, and we want to solve the following equation for x:

$$E = \frac{n}{n+p} = \frac{xn}{xn+kp}.$$

Well, if $x = k$, there will be a common factor of k in the denominator $xn + kp = kn + kp = k(n+p)$, and we can reduce the fraction to get

$$\frac{xn}{xn+kp} = \frac{kn}{k(n+p)} = \frac{n}{n+p}.$$

In other words, if we increase the problem size at the same rate that we increase the number of processes/threads, then the efficiency will be unchanged, and our program is scalable.

There are a couple of cases that have special names. If when we increase the number of processes/threads, we can keep the efficiency fixed *without* increasing the problem size, the program is said to be *strongly scalable.* If we can keep the efficiency fixed by increasing the problem size at the same rate as we increase the number of processes/threads, then the program is said to be *weakly scalable.* The program in our example would be weakly scalable.

2.6.4 Taking timings

You may have been wondering how we find T_{serial} and T_{parallel}. There are a *lot* of different approaches, and with parallel programs the details may depend on the API. However, there are a few general observations we can make that may make things a little easier.

The first thing to note is that there are at least two different reasons for taking timings. During program development we may take timings in order to determine if the program is behaving as we intend. For example, in a distributed-memory program we might be interested in finding out how much time the processes are spending waiting for messages, because if this value is large, there is almost certainly something wrong either with our design or our implementation. On the other hand, once we've completed development of the program, we're often interested in determining how good its performance is. Perhaps surprisingly, the way we take these two timings is usually different. For the first timing, we usually need very detailed information: How much time did the program spend in this part of the program? How much time did it spend in that part? For the second, we usually report a single value. Right now we'll talk about the second type of timing. See Exercise 2.22 for a brief discussion of some issues in taking the first type of timing.

Second, we're usually *not* interested in the time that elapses between the program's start and the program's finish. We're usually interested only in some part of the program. For example, if we write a program that implements bubble sort, we're probably only interested in the time it takes to sort the keys, not the time it takes to read them in and print them out. We probably can't use something like the Unix shell command `time`, which reports the time taken to run a program from start to finish.

Third, we're usually *not* interested in "CPU time." This is the time reported by the standard C function `clock`. It's the total time the program spends in code executed as part of the program. It would include the time for code we've written; it would include the time we spend in library functions such as `pow` or `sin`; and it would include the time the operating system spends in functions we call, such as `printf` and `scanf`. It would not include time the program was idle, and this could be a problem. For example, in a distributed-memory program, a process that calls a receive function may have to wait for the sending process to execute the matching send, and the operating system might put the receiving process to sleep while it waits. This idle time wouldn't be counted as CPU time, since no function that's been called by the process is active. However, it should count in our evaluation of the overall run-time, since it may be a real cost in our program. If each time the program is run, the process has to wait, ignoring the time it spends waiting would give a misleading picture of the actual run-time of the program.

Thus, when you see an article reporting the run-time of a parallel program, the reported time is usually "wall clock" time. That is, the authors of the article report the time that has elapsed between the start and finish of execution of the code that the user is interested in. If the user could see the execution of the program, she would

hit the start button on her stopwatch when it begins execution and hit the stop button when it stops execution. Of course, she can't see her code executing, but she can modify the source code so that it looks something like this:

```
double start, finish;
. . .
start = Get_current_time();
/* Code that we want to time */
. . .
finish = Get_current_time();
printf("The elapsed time = %e seconds\n", finish-start);
```

The function `Get_current_time()` is a hypothetical function that's supposed to return the number of seconds that have elapsed since some fixed time in the past. It's just a placeholder. The actual function that is used will depend on the API. For example, MPI has a function `MPI_Wtime` that could be used here, and the OpenMP API for shared-memory programming has a function `omp_get_wtime`. Both functions return wall clock time instead of CPU time.

There may be an issue with the **resolution** of the timer function. The resolution is the unit of measurement on the timer. It's the duration of the shortest event that can have a nonzero time. Some timer functions have resolutions in milliseconds (10^{-3} seconds), and when instructions can take times that are less than a nanosecond (10^{-9} seconds), a program may have to execute millions of instructions before the timer reports a nonzero time. Many APIs provide a function that reports the resolution of the timer. Other APIs specify that a timer must have a given resolution. In either case we, as the programmers, need to check these values.

When we're timing parallel programs, we need to be a little more careful about how the timings are taken. In our example, the code that we want to time is probably being executed by multiple processes or threads and our original timing will result in the output of p elapsed times.

```
private double start, finish;
. . .
start = Get_current_time();
/* Code that we want to time */
. . .
finish = Get_current_time();
printf("The elapsed time = %e seconds\n", finish-start);
```

However, what we're usually interested in is a single time: the time that has elapsed from when the first process/thread began execution of the code to the time the last process/thread finished execution of the code. We often can't obtain this exactly, since there may not be any correspondence between the clock on one node and the clock on another node. We usually settle for a compromise that looks something like this:

```
shared double global_elapsed;
private double my_start, my_finish, my_elapsed;
. . .
```

```
/* Synchronize all processes/threads */
Barrier();
my_start = Get_current_time();

/* Code that we want to time */
. . .

my_finish = Get_current_time();
my_elapsed = my_finish - my_start;

/* Find the max across all processes/threads */
global_elapsed = Global_max(my_elapsed);
if (my_rank == 0)
   printf("The elapsed time = %e seconds\n", global_elapsed);
```

Here, we first execute a **barrier** function that approximately synchronizes all of the processes/threads. We would like for all the processes/threads to return from the call simultaneously, but such a function usually can only guarantee that all the processes/threads have started the call when the first process/thread returns. We then execute the code as before and each process/thread finds the time it took. Then all the processes/threads call a global maximum function, which returns the largest of the elapsed times, and process/thread 0 prints it out.

We also need to be aware of the *variability* in timings. When we run a program several times, it's extremely likely that the elapsed time will be different for each run. This will be true even if each time we run the program we use the same input and the same systems. It might seem that the best way to deal with this would be to report either a mean or a median run-time. However, it's unlikely that some outside event could actually make our program run faster than its best possible run-time. So instead of reporting the mean or median time, we usually report the *minimum* time.

Running more than one thread per core can cause dramatic increases in the variability of timings. More importantly, if we run more than one thread per core, the system will have to take extra time to schedule and deschedule cores, and this will add to the overall run-time. Therefore, we rarely run more than one thread per core.

Finally, as a practical matter, since our programs won't be designed for high-performance I/O, we'll usually not include I/O in our reported run-times.

2.7 PARALLEL PROGRAM DESIGN

So we've got a serial program. How do we parallelize it? We know that in general we need to divide the work among the processes/threads so that each process gets roughly the same amount of work and communication is minimized. In most cases, we also need to arrange for the processes/threads to synchronize and communicate.

Unfortunately, there isn't some mechanical process we can follow; if there were, we could write a program that would convert any serial program into a parallel program, but, as we noted in Chapter 1, in spite of a tremendous amount of work and some progress, this seems to be a problem that has no universal solution.

However, Ian Foster provides an outline of steps in his online book *Designing and Building Parallel Programs* [19]:

1. *Partitioning*. Divide the computation to be performed and the data operated on by the computation into small tasks. The focus here should be on identifying tasks that can be executed in parallel.
2. *Communication*. Determine what communication needs to be carried out among the tasks identified in the previous step.
3. *Agglomeration or aggregation*. Combine tasks and communications identified in the first step into larger tasks. For example, if task A must be executed before task B can be executed, it may make sense to aggregate them into a single composite task.
4. *Mapping*. Assign the composite tasks identified in the previous step to processes/threads. This should be done so that communication is minimized, and each process/thread gets roughly the same amount of work.

This is sometimes called **Foster's methodology**.

2.7.1 An example

Let's look at a small example. Suppose we have a program that generates large quantities of floating point data that it stores in an array. In order to get some feel for the distribution of the data, we can make a histogram of the data. Recall that to make a histogram, we simply divide the range of the data up into equal sized subintervals, or *bins*, determine the number of measurements in each bin, and plot a bar graph showing the relative sizes of the bins. As a very small example, suppose our data are

$$1.3, 2.9, 0.4, 0.3, 1.3, 4.4, 1.7, 0.4, 3.2, 0.3, 4.9, 2.4, 3.1, 4.4, 3.9, 0.4, 4.2, 4.5, 4.9, 0.9.$$

Then the data lie in the range 0–5, and if we choose to have five bins, the histogram might look something like Figure 2.20.

A serial program

It's pretty straightforward to write a serial program that generates a histogram. We need to decide what the bins are, determine the number of measurements in each bin, and print the bars of the histogram. Since we're not focusing on I/O, we'll limit ourselves to just the first two steps, so the input will be

1. the number of measurements, `data_count`;
2. an array of `data_count` floats, `data`;
3. the minimum value for the bin containing the smallest values, `min_meas`;
4. the maximum value for the bin containing the largest values, `max_meas`;
5. the number of bins, `bin_count`;

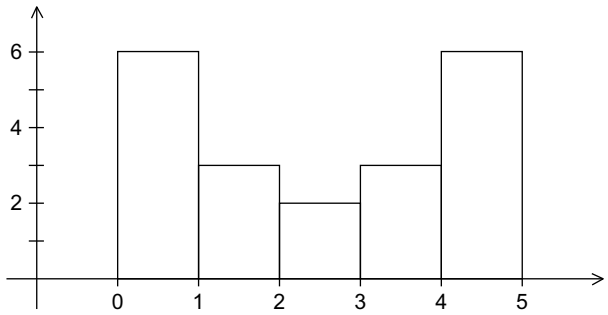


FIGURE 2.20

A histogram

The output will be an array containing the number of elements of data that lie in each bin. To make things precise, we'll make use of the following data structures:

- `bin_maxes`. An array of `bin_count` floats
- `bin_counts`. An array of `bin_count` ints

The array `bin_maxes` will store the upper bound for each bin, and `bin_counts` will store the number of data elements in each bin. To be explicit, we can define

```
bin_width = (max_meas – min_meas)/bin_count
```

Then `bin_maxes` will be initialized by

```
for (b = 0; b < bin_count; b++)
   bin_maxes[b] = min_meas + bin_width*(b+1);
```

We'll adopt the convention that bin b will be all the measurements in the range

```
bin_maxes[b–1] <= measurement < bin_maxes[b]
```

Of course, this doesn't make sense if $b = 0$, and in this case we'll use the rule that bin 0 will be the measurements in the range

```
min_meas <= measurement < bin_maxes[0]
```

This means we always need to treat bin 0 as a special case, but this isn't too onerous.

Once we've initialized `bin_maxes` and assigned 0 to all the elements of `bin_counts`, we can get the counts by using the following pseudo-code:

```
for (i = 0; i < data_count; i++) {
   bin = Find_bin(data[i], bin_maxes, bin_count, min_meas);
   bin_counts[bin]++;
}
```

The `Find_bin` function returns the bin that `data[i]` belongs to. This could be a simple linear search function: search through `bin_maxes` until you find a bin b that satisfies

```
bin_maxes[b–1] <= data[i] < bin_maxes[b]
```

(Here we're thinking of `bin_maxes[-1]` as `min_meas`.) This will be fine if there aren't very many bins, but if there are a lot of bins, binary search will be much better.

Parallelizing the serial program

If we assume that `data_count` is much larger than `bin_count`, then even if we use binary search in the `Find_bin` function, the vast majority of the work in this code will be in the loop that determines the values in `bin_counts`. The focus of our parallelization should therefore be on this loop, and we'll apply Foster's methodology to it. The first thing to note is that the outcomes of the steps in Foster's methodology are by no means uniquely determined, so you shouldn't be surprised if at any stage you come up with something different.

For the first step we might identify two types of tasks: finding the bin to which an element of `data` belongs and incrementing the appropriate entry in `bin_counts`.

For the second step, there must be a communication between the computation of the appropriate bin and incrementing an element of `bin_counts`. If we represent our tasks with ovals and communications with arrows, we'll get a diagram that looks something like that shown in Figure 2.21. Here, the task labelled with "`data[i]`" is determining which bin the value `data[i]` belongs to, and the task labelled with "`bin_counts[b]++`" is incrementing `bin_counts[b]`.

For any fixed element of `data`, the tasks "find the bin `b` for element of `data`" and "increment `bin_counts[b]`" can be aggregated, since the second can only happen once the first has been completed.

However, when we proceed to the final or mapping step, we see that if two processes or threads are assigned elements of `data` that belong to the same bin `b`, they'll both result in execution of the statement `bin_counts[b]++`. If `bin_counts[b]` is shared (e.g., the array `bin_counts` is stored in shared-memory), then this will result in a race condition. If `bin_counts` has been partitioned among the processes/threads, then updates to its elements will require communication. An alternative is to store multiple "local" copies of `bin_counts` and add the values in the local copies after all the calls to `Find_bin`.

If the number of bins, `bin_count`, isn't absolutely gigantic, there shouldn't be a problem with this. So let's pursue this alternative, since it is suitable for use on both shared- and distributed-memory systems.

In this setting, we need to update our diagram so that the second collection of tasks increments `loc_bin_cts[b]`. We also need to add a third collection of tasks, adding the various `loc_bin_cts[b]` to get `bin_counts[b]`. See Figure 2.22. Now we

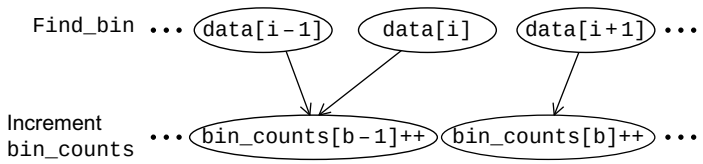


FIGURE 2.21

The first two stages of Foster's methodology

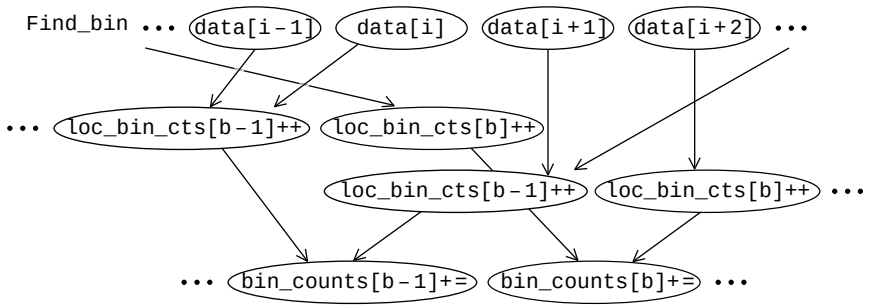


FIGURE 2.22

Alternative definition of tasks and communication

see that if we create an array `loc_bin_cts` for each process/thread, then we can map the tasks in the first two groups as follows:

1. Elements of `data` are assigned to the processes/threads so that each process/thread gets roughly the same number of elements.
2. Each process/thread is responsible for updating its `loc_bin_cts` array on the basis of its assigned elements.

To finish up, we need to add the elements `loc_bin_cts[b]` into `bin_counts[b]`. If both the number of processes/threads is small and the number of bins is small, all of the additions can be assigned to a single process/thread. If the number of bins is much larger than the number of processes/threads, we can divide the bins among the processes/threads in much the same way that we divided the elements of `data`. If the number of processes/threads is large, we can use a tree-structured global sum similar to the one we discussed in Chapter 1. The only difference is that now the sending process/threads are sending an array, and the receiving process/threads are receiving and adding an array. Figure 2.23 shows an example with eight processes/threads. Each

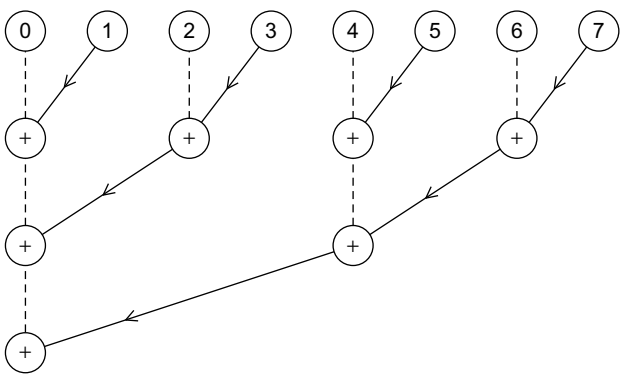


FIGURE 2.23

Adding the local arrays

circle in the top row corresponds to a process/thread. Between the first and the second rows, the odd-numbered processes/threads make their `loc_bin_cts` available to the even-numbered processes/threads. Then in the second row, the even-numbered processes/threads add the new counts to their existing counts. Between the second and third rows the process is repeated with the processes/threads whose ranks aren't divisible by four sending to those whose are. This process is repeated until process/thread 0 has computed `bin_counts`.

2.8 WRITING AND RUNNING PARALLEL PROGRAMS

In the past, virtually all parallel program development was done using a text editor such as `vi` or `Emacs`, and the program was either compiled and run from the command line or from within the editor. Debuggers were also typically started from the command line. Now there are also integrated development environments (IDEs) available from Microsoft, the Eclipse project, and others; see [16, 38].

In smaller shared-memory systems, there is a single running copy of the operating system, which ordinarily schedules the threads on the available cores. On these systems, shared-memory programs can usually be started using either an IDE or the command line. Once started, the program will typically use the console and the keyboard for input from `stdin` and output to `stdout` and `stderr`. On larger systems, there may be a batch scheduler, that is, a user requests a certain number of cores, and specifies the path to the executable and where input and output should go (typically to files in secondary storage).

In typical distributed-memory and hybrid systems, there is a host computer that is responsible for allocating nodes among the users. Some systems are purely *batch* systems, which are similar to shared-memory batch systems. Others allow users to check out nodes and run jobs interactively. Since job startup often involves communicating with remote systems, the actual startup is usually done with a script. For example, MPI programs are usually started with a script called `mpirun` or `mpiexec`.

As usual, RTFD, which is sometimes translated as "read the fine documentation."

2.9 ASSUMPTIONS

As we noted earlier, we'll be focusing on homogeneous MIMD systems—systems in which all of the nodes have the same architecture—and our programs will be SPMD. Thus, we'll write a single program that can use branching to have multiple different behaviors. We'll assume the cores are identical but that they operate asynchronously. We'll also assume that we always run at most one process or thread of our program on a single core, and we'll often use static processes or threads. In other words, we'll often start all of our processes or threads at more or less the same time, and when they're done executing, we'll terminate them at more or less the same time.

Some application programming interfaces or APIs for parallel systems define new programming languages. However, most extend existing languages, either through a library of functions—for example, functions to pass messages—or extensions to the compiler for the serial language. This latter approach will be the focus of this text. We'll be using parallel extensions to the C language.

When we want to be explicit about compiling and running programs, we'll use the command line of a Unix shell, the `gcc` compiler or some extension of it (e.g., `mpicc`), and we'll start programs from the command line. For example, if we wanted to show compilation and execution of the "hello, world" program from Kernighan and Ritchie [29], we might show something like this:

```
$ gcc -g -Wall -o hello hello.c
$ ./hello
hello, world
```

The $-sign is the prompt from the shell. We will usually use the following options for the compiler:

- `-g`. Create information that allows us to use a debugger
- `-Wall`. Issue lots of warnings
- `-o <outfile>`. Put the executable in the file named `outfile`
- When we're timing programs, we usually tell the compiler to optimize the code by using the `-O2` option.

In most systems, user directories or folders are not, by default, in the user's execution path, so we'll usually start jobs by giving the path to the executable by adding `./` to its name.

2.10 SUMMARY

There's a *lot* of material in this chapter, and a complete summary would run on for many pages, so we'll be very terse.

2.10.1 Serial systems

We started with a discussion of conventional serial hardware and software. The standard model of computer hardware has been the **von Neumann architecture**, which consists of a **central processing unit** that carries out the computations, and **main memory** that stores data and instructions. The separation of CPU and memory is often called the **von Neumann bottleneck** since it limits the rate at which instructions can be executed.

Perhaps the most important software on a computer is the **operating system**. It manages the computer's resources. Most modern operating systems are **multitasking**. Even if the hardware doesn't have multiple processors or cores, by rapidly switching among executing programs, the OS creates the illusion that multiple jobs are running simultaneously. A running program is called a **process**. Since a running

process is more or less autonomous, it has a lot of associated data and information. A **thread** is started by a process. A thread doesn't need as much associated data and information, and threads can be stopped and started much faster than processes.

In computer science, **caches** are memory locations that can be accessed faster than other memory locations. CPU caches are memory locations that are intermediate between the CPU registers and main memory. Their purpose is to reduce the delays associated with accessing main memory. Data are stored in cache using the principle of locality, that is, items that are close to recently accessed items are more likely to be accessed in the near future. Thus, contiguous **blocks** or **lines** of data and instructions are transferred between main memory and caches. When an instruction or data item is accessed and it's already in cache, it's called a cache **hit**. If the item isn't in cache, it's called a cache **miss**. Caches are directly managed by the computer hardware, so programmers can only indirectly control caching.

Main memory can also function as a cache for secondary storage. This is managed by the hardware and the operating system through **virtual memory**. Rather than storing all of a program's instructions and data in main memory, only the active parts are stored in main memory, and the remaining parts are stored in secondary storage called **swap space**. Like CPU caches, virtual memory operates on blocks of contiguous data and instructions, which in this setting are called **pages**. Note that instead of addressing the memory used by a program with physical addresses, virtual memory uses **virtual addresses**, which are independent of actual physical addresses. The correspondence between physical addresses and virtual addresses is stored in main memory in a **page table**. The combination of virtual addresses and a page table provides the system with the flexibility to store a program's data and instructions anywhere in memory. Thus, it won't matter if two different programs use the same virtual addresses. With the page table stored in main memory, it could happen that every time a program needed to access a main memory location it would need two memory accesses: one to get the appropriate page table entry so it could find the location in main memory, and one to actually access the desired memory. In order to avoid this problem, CPUs have a special page table cache called the **translation lookaside buffer**, which stores the most recently used page table entries.

Instruction-level parallelism (ILP) allows a single processor to execute multiple instructions simultaneously. There are two main types of ILP: **pipelining** and **multiple issue**. With pipelining, some of the functional units of a processor are ordered in sequence, with the output of one being the input to the next. Thus, while one piece of data is being processed by, say, the second functional unit, another piece of data can be processed by the first. With multiple issue, the functional units are replicated, and the processor attempts to simultaneously execute different instructions in a single program.

Rather than attempting to simultaneously execute individual instructions, **hardware multithreading** attempts to simultaneously execute different threads. There are several different approaches to implementing hardware multithreading. However, all of them attempt to keep the processor as busy as possible by rapidly switching

between threads. This is especially important when a thread **stalls** and has to wait (e.g., for a memory access to complete) before it can execute an instruction. In **simultaneous multithreading**, the different threads can simultaneously use the multiple functional units in a multiple issue processor. Since threads consist of multiple instructions, we sometimes say that **thread-level paralleism**, or TLP, is **coarser-grained** than ILP.

2.10.2 Parallel hardware

ILP and TLP provide parallelism at a very low level; they're typically controlled by the processor and the operating system, and their use isn't directly controlled by the programmer. For our purposes, hardware is parallel if the parallelism is visible to the programmer, and she can modify her source code to exploit it.

Parallel hardware is often classified using **Flynn's taxonomy**, which distinguishes between the number of instruction streams and the number of data streams a system can handle. A von Neumann system has a single instruction stream and a single data stream so it is classified as a **single instruction, single data**, or **SISD**, system.

A **single instruction, multiple data**, or **SIMD**, system executes a single instruction at a time, but the instruction can operate on multiple data items. These systems often execute their instructions in lockstep: the first instruction is applied to all of the data elements simultaneously, then the second is applied, and so on. This type of parallel system is usually employed in **data parallel** programs, programs in which the data are divided among the processors and each data item is subjected to more or less the same sequence of instructions. **Vector processors** and **graphics processing units** are often classified as SIMD systems, although current generation GPUs also have characteristics of multiple instruction, multiple data stream systems.

Branching in SIMD systems is handled by idling those processors that might operate on a data item to which the instruction doesn't apply. This behavior makes SIMD systems poorly suited for **task-parallelism**, in which each processor executes a different task, or even data-parallelism, with many conditional branches.

As the name suggests, **multiple instruction, multiple data**, or **MIMD**, systems execute multiple independent instruction streams, each of which can have its own data stream. In practice, MIMD systems are collections of autonomous processors that can execute at their own pace. The principal distinction between different MIMD systems is whether they are **shared-memory** or **distributed-memory** systems. In shared-memory systems, each processor or core can directly access every memory location, while in distributed-memory systems, each processor has its own private memory. Most of the larger MIMD systems are **hybrid** systems in which a number of relatively small shared-memory systems are connected by an interconnection network. In such systems, the individual shared-memory systems are sometimes called **nodes**. Some MIMD systems are **heterogeneous** systems, in which the processors have different capabilities. For example, a system with a conventional CPU and a GPU is a heterogeneous system. A system in which all the processors have the same architecture is **homogeneous**.

There are a number of different interconnects for joining processors to memory in shared-memory systems and for interconnecting the processors in distributed-memory or hybrid systems. The most commonly used interconnects for shared-memory are **buses** and **crossbars**. Distributed-memory systems sometimes use **direct** interconnects such as **toroidal meshes** and **hypercubes**, and they sometimes use **indirect** interconnects such as crossbars and **multistage** networks. Networks are often evaluated by examining the **bisection width** or the **bisection bandwidth** of the network. These give measures of how much simultaneous communication the network can support. For individual communications between nodes, authors often discuss the **latency** and **bandwidth** of the interconnect.

A potential problem with shared-memory systems is **cache coherence**. The same variable can be stored in the caches of two different cores, and if one core updates the value of the variable, the other core may be unaware of the change. There are two main methods for insuring cache coherence: **snooping** and the use of **directories**. Snooping relies on the capability of the interconnect to broadcast information from each cache controller to every other cache controller. Directories are special distributed data structures, which store information on each cache line. Cache coherence introduces another problem for shared-memory programming: **false sharing**. When one core updates a variable in one cache line, and another core wants to access *another* variable in the same cache line, it will have to access main memory, since the unit of cache coherence is the cache line. That is, the second core only "knows" that the line it wants to access has been updated. It doesn't know that the variable it wants to access hasn't been changed.

2.10.3 Parallel software

In this text we'll focus on developing software for homogeneous MIMD systems. Most programs for such systems consist of a single program that obtains parallelism by branching. Such programs are often called **single program, multiple data** or **SPMD** programs. In shared-memory programs we'll call the instances of running tasks threads; in distributed-memory programs we'll call them processes.

Unless our problem is **embarrassingly parallel**, the development of a parallel program needs at a minimum to address the issues of **load balance**, **communication**, and **synchronization** among the processes or threads.

In shared-memory programs, the individual threads can have **private** and **shared-memory**. Communication is usually done through **shared variables**. Any time the processors execute asynchronously, there is the potential for **nondeterminism**, that is, for a given input the behavior of the program can change from one run to the next. This can be a serious problem, especially in shared-memory programs. If the nondeterminism results from two threads' attempts to access the same resource, and it can result in an error, the program is said to have a **race condition**. The most common place for a race condition is a **critical section**, a block of code that can only be executed by one thread at a time. In most shared-memory APIs, **mutual exclusion**

in a critical section can be enforced with an object called a **mutual exclusion lock** or **mutex**. Critical sections should be made as small as possible, since a mutex will allow only one thread at a time to execute the code in the critical section, effectively making the code serial.

A second potential problem with shared-memory programs is **thread safety**. A block of code that functions correctly when it is run by multiple threads is said to be **thread safe**. Functions that were written for use in serial programs can make unwitting use of shared data—for example, static variables—and this use in a multithreaded program can cause errors. Such functions are not thread safe.

The most common API for programming distributed-memory systems is **message-passing**. In message-passing, there are (at least) two distinct functions: a **send** function and a **receive** function. When processes need to communicate, one calls the send and the other calls the receive. There are a variety of possible behaviors for these functions. For example, the send can **block** or wait until the matching receive has started, or the message-passing software can can copy the data for the message into its own storage, and the sending process can return before the matching receive has started. The most common behavior for receives is to block until the message has been received. The most commonly used message-passing system is called the **Message-Passing Interface** or MPI. It provides a great deal of functionality beyond simple sends and receives.

Distributed-memory systems can also be programmed using **one-sided communications**, which provide functions for accessing memory belonging to another process, and **partitioned global address space** languages, which provide some shared-memory functionality in distributed-memory systems.

2.10.4 Input and output

In general parallel systems, multiple cores can access multiple secondary storage devices. We won't attempt to write programs that make use of this functionality. Rather, we'll write programs in which one process or thread can access `stdin`, and all processes can access `stdout` and `stderr`. However, because of nondeterminism, except for debug output we'll usually have a single process or thread accessing `stdout`.

2.10.5 Performance

If we run a parallel program with p processes or threads and no more than one process/thread per core, then our ideal would be for our parallel program to run p times faster than the serial program. This is called **linear speedup**, but in practice it is rarely achieved. If we denote the run-time of the serial program by T_{serial} and the parallel program's run-time by T_{parallel}, then the **speedup** S and **efficiency** E of the parallel program are given by the formulas

$$S = \frac{T_{\text{serial}}}{T_{\text{parallel}}}, \text{ and } E = \frac{T_{\text{serial}}}{pT_{\text{parallel}}},$$

respectively. So linear speedup has $S = p$ and $E = 1$. In practice, we will almost always have $S < p$ and $E < 1$. If we fix the problem size, E usually decreases as we increase p, while if we fix the number of processes/threads, then S and E often increase as we increase the problem size.

Amdahl's law provides an upper bound on the speedup that can be obtained by a parallel program: if a fraction r of the original, serial program isn't parallelized, then we can't possibly get a speedup better than $1/r$, regardless of how many processes/threads we use. In practice many parallel programs obtain excellent speedups. One possible reason for this apparent contradiction is that Amdahl's law doesn't take into consideration the fact that the unparallelized part often decreases in size relative to the parallelized part as the problem size increases.

Scalability is a term that has many interpretations. In general, a technology is scalable if it can handle ever-increasing problem sizes. Formally, a parallel program is scalable if there is a rate at which the problem size can be increased so that as the number of processes/threads is increased, the efficiency remains constant. A program is **strongly** scalable, if the problem size can remain fixed, and it's **weakly** scalable if the problem size needs to be increased at the same rate as the number of processes/threads.

In order to determine T_{serial} and T_{parallel}, we usually need to include calls to a timer function in our source code. We want these timer functions to give **wall clock** time, not CPU time, since the program may be "active"—for example, waiting for a message—even when the CPU is idle. To take parallel times, we usually want to synchronize the processes/threads before starting the timer, and, after stopping the timer, find the maximum elapsed time among all the processes/threads. Because of system variability, we usually need to run a program several times with a given data set, and we usually take the minimum time from the multiple runs. To reduce variability and improve overall run-times, we usually run no more than one thread per core.

2.10.6 Parallel program design

Foster's methodology provides a sequence of steps that can be used to design parallel programs. The steps are **partitioning** the problem to identify tasks, identifying **communication** among the tasks, **agglomeration** or **aggregation** to group tasks, and **mapping** to assign aggregate tasks to processes/threads.

2.10.7 Assumptions

We'll be focusing on the development of parallel programs for both shared- and distributed-memory MIMD systems. We'll write SPMD programs that usually use **static** processes or threads—processes/threads that are created when the program begins execution, and are not shut down until the program terminates. We'll also assume that we run at most one process or thread on each core of the system.

2.11 EXERCISES

2.1. When we were discussing floating point addition, we made the simplifying assumption that each of the functional units took the same amount of time. Suppose that fetch and store each take 2 nanoseconds and the remaining operations each take 1 nanosecond.
- **a.** How long does a floating point addition take with these assumptions?
- **b.** How long will an unpipelined addition of 1000 pairs of floats take with these assumptions?
- **c.** How long will a pipelined addition of 1000 pairs of floats take with these assumptions?
- **d.** The time required for fetch and store may vary considerably if the operands/results are stored in different levels of the memory hierarchy. Suppose that a fetch from a level 1 cache takes two nanoseconds, while a fetch from a level 2 cache takes five nanoseconds, and a fetch from main memory takes fifty nanoseconds. What happens to the pipeline when there is a level 1 cache miss on a fetch of one of the operands? What happens when there is a level 2 miss?

2.2. Explain how a queue, implemented in hardware in the CPU, could be used to improve the performance of a write-through cache.

2.3. Recall the example involving cache reads of a two-dimensional array (page 22). How does a larger matrix and a larger cache affect the performance of the two pairs of nested loops? What happens if `MAX = 8` and the cache can store four lines? How many misses occur in the reads of `A` in the first pair of nested loops? How many misses occur in the second pair?

2.4. In Table 2.2, virtual addresses consist of a byte offset of 12 bits and a virtual page number of 20 bits. How many pages can a program have if it's run on a system with this page size and this virtual address size?

2.5. Does the addition of cache and virtual memory to a von Neumann system change its designation as an SISD system? What about the addition of pipelining? Multiple issue? Hardware multithreading?

2.6. Suppose that a vector processor has a memory system in which it takes 10 cycles to load a single 64-bit word from memory. How many memory banks are needed so that a stream of loads can, on average, require only one cycle per load?

2.7. Discuss the differences in how a GPU and a vector processor might execute the following code:

```
sum = 0.0;
for (i = 0; i < n; i++) {
   y[i] += a*x[i];
   sum += z[i]*z[i];
}
```

2.8. Explain why the performance of a hardware multithreaded processing core might degrade if it had large caches and it ran many threads.

2.9. In our discussion of parallel hardware, we used Flynn's taxonomy to identify three types of parallel systems: SISD, SIMD, and MIMD. None of our systems were identified as multiple instruction, single data, or MISD. How would an MISD system work? Give an example.

2.10. Suppose a program must execute 10^{12} instructions in order to solve a particular problem. Suppose further that a single processor system can solve the problem in 10^6 seconds (about 11.6 days). So, on average, the single processor system executes 10^6 or a million instructions per second. Now suppose that the program has been parallelized for execution on a distributed-memory system. Suppose also that if the parallel program uses p processors, each processor will execute $10^{12}/p$ instructions and each processor must send $10^9(p-1)$ messages. Finally, suppose that there is no additional overhead in executing the parallel program. That is, the program will complete after each processor has executed all of its instructions and sent all of its messages, and there won't be any delays due to things such as waiting for messages.

- **a.** Suppose it takes 10^{-9} seconds to send a message. How long will it take the program to run with 1000 processors, if each processor is as fast as the single processor on which the serial program was run?
- **b.** Suppose it takes 10^{-3} seconds to send a message. How long will it take the program to run with 1000 processors?

2.11. Derive formulas for the total number of links in the various distributed-memory interconnects.

2.12.

- **a.** A planar mesh is just like a toroidal mesh, except that it doesn't have the "wraparound" links. What is the bisection width of a square planar mesh?
- **b.** A three-dimensional mesh is similar to a planar mesh, except that it also has depth. What is the bisection width of a three-dimensional mesh?

2.13.

- **a.** Sketch a four-dimensional hypercube.
- **b.** Use the inductive definition of a hypercube to explain why the bisection width of a hypercube is $p/2$.

2.14. To define the bisection width for indirect networks, the processors are partitioned into two groups so that each group has half the processors. Then, links are removed from *anywhere* in the network so that the two groups are no longer connected. The minimum number of links removed is the bisection width. When we count links, if the diagram uses unidirectional links, two unidirectional links count as one link. Show that an eight-by-eight crossbar has a bisection width less than or equal to eight. Also show that an omega network with eight processors has a bisection width less than or equal to four.

2.15. **a.** Suppose a shared-memory system uses snooping cache coherence and write-back caches. Also suppose that core 0 has the variable `x` in its cache, and it executes the assignment `x = 5`. Finally suppose that core 1 doesn't have `x` in its cache, and after core 0's update to `x`, core 1 tries to execute `y = x`. What value will be assigned to `y`? Why?

b. Suppose that the shared-memory system in the previous part uses a directory-based protocol. What value will be assigned to `y`? Why?

c. Can you suggest how any problems you found in the first two parts might be solved?

2.16. **a.** Suppose the run-time of a serial program is given by $T_{\text{serial}} = n^2$, where the units of the run-time are in microseconds. Suppose that a parallelization of this program has run-time $T_{\text{parallel}} = n^2/p + \log_2(p)$. Write a program that finds the speedups and efficiencies of this program for various values of n and p. Run your program with $n = 10, 20, 40, \ldots, 320$, and $p = 1, 2, 4, \ldots, 128$. What happens to the speedups and efficiencies as p is increased and n is held fixed? What happens when p is fixed and n is increased?

b. Suppose that $T_{\text{parallel}} = T_{\text{serial}}/p + T_{\text{overhead}}$. Also suppose that we fix p and increase the problem size.

- Show that if T_{overhead} grows more slowly than T_{serial}, the parallel efficiency will increase as we increase the problem size.
- Show that if, on the other hand, T_{overhead} grows faster than T_{serial}, the parallel efficiency will decrease as we increase the problem size.

2.17. A parallel program that obtains a speedup greater than p—the number of processes or threads—is sometimes said to have **superlinear speedup**. However, many authors don't count programs that overcome "resource limitations" as having superlinear speedup. For example, a program that must use secondary storage for its data when it's run on a single processor system might be able to fit all its data into main memory when run on a large distributed-memory system. Give another example of how a program might overcome a resource limitation and obtain speedups greater than p.

2.18. Look at three programs you wrote in your Introduction to Computer Science class. What (if any) parts of these programs are inherently serial? Does the inherently serial part of the work done by the program decrease as the problem size increases? Or does it remain roughly the same?

2.19. Suppose $T_{\text{serial}} = n$ and $T_{\text{parallel}} = n/p + \log_2(p)$, where times are in microseconds. If we increase p by a factor of k, find a formula for how much we'll need to increase n in order to maintain constant efficiency. How much should we increase n by if we double the number of processes from 8 to 16? Is the parallel program scalable?

2.20. Is a program that obtains linear speedup strongly scalable? Explain your answer.

2.21. Bob has a program that he wants to time with two sets of data, `input_data1` and `input_data2`. To get some idea of what to expect before adding timing functions to the code he's interested in, he runs the program with two sets of data and the Unix shell command `time`:

```
$ time ./bobs_prog < input_data1

real 0m0.001s
user 0m0.001s
sys  0m0.000s
$ time ./bobs_prog < input_data2

real 1m1.234s
user 1m0.001s
sys  0m0.111s
```

The timer function Bob is using has millisecond resolution. Should Bob use it to time his program with the first set of data? What about the second set of data? Why or why not?

2.22. As we saw in the preceding problem, the Unix shell command `time` reports the user time, the system time, and the "real" time or total elapsed time. Suppose that Bob has defined the following functions that can be called in a C program:

```
double utime(void);
double stime(void);
double rtime(void);
```

The first returns the number of seconds of user time that have elapsed since the program started execution, the second returns the number of system seconds, and the third returns the total number of seconds. Roughly, user time is time spent in the user code and library functions that don't need to use the operating system—for example, `sin` and `cos`. System time is time spent in functions that do need to use the operating system—for example, `printf` and `scanf`.

a. What is the mathematical relation among the three function values? That is, suppose the program contains the following code:

```
u = double utime(void);
s = double stime(void);
r = double rtime(void);
```

Write a formula that expresses the relation between `u`, `s`, and `r`. (You can assume that the time it takes to call the functions is negligible.)

b. On Bob's system, any time that an MPI process spends waiting for messages isn't counted by either `utime` or `stime`, but the time *is* counted by `rtime`. Explain how Bob can use these facts to determine whether an MPI process is spending too much time waiting for messages.

c. Bob has given Sally his timing functions. However, Sally has discovered that on her system, the time an MPI process spends waiting for messages is counted as user time. Furthermore, sending messages doesn't use any system time. Can Sally use Bob's functions to determine whether an MPI process is spending too much time waiting for messages? Explain your answer.

2.23. In our application of Foster's methodology to the construction of a histogram, we essentially identified aggregate tasks with elements of `data`. An apparent alternative would be to identify aggregate tasks with elements of `bin_counts`, so an aggregate task would consist of all increments of `bin_counts[b]` and consequently all calls to `Find_bin` that return `b`. Explain why this aggregation might be a problem.

2.24. If you haven't already done so in Chapter 1, try to write pseudo-code for our tree-structured global sum, which sums the elements of `loc_bin_cts`. First consider how this might be done in a shared-memory setting. Then consider how this might be done in a distributed-memory setting. In the shared-memory setting, which variables are shared and which are private?

This page intentionally left blank

CHAPTER

Distributed-Memory Programming with MPI

3

Recall that the world of parallel multiple instruction, multiple data, or MIMD, computers is, for the most part, divided into **distributed-memory** and **shared-memory** systems. From a programmer's point of view, a distributed-memory system consists of a collection of core-memory pairs connected by a network, and the memory associated with a core is directly accessible only to that core. See Figure 3.1. On the other hand, from a programmer's point of view, a shared-memory system consists of a collection of cores connected to a globally accessible memory, in which each core can have access to any memory location. See Figure 3.2. In this chapter we're going to start looking at how to program distributed-memory systems using **message-passing**.

Recall that in message-passing programs, a program running on one core-memory pair is usually called a **process**, and two processes can communicate by calling functions: one process calls a *send* function and the other calls a *receive* function. The implementation of message-passing that we'll be using is called **MPI**, which is an abbreviation of **Message-Passing Interface**. MPI is not a new programming language. It defines a *library* of functions that can be called from C, C++, and Fortran programs. We'll learn about some of MPI's different send and receive functions. We'll also learn about some "global" communication functions that can involve more than two processes. These functions are called **collective** communications. In the process of learning about all of these MPI functions, we'll also learn about some of the

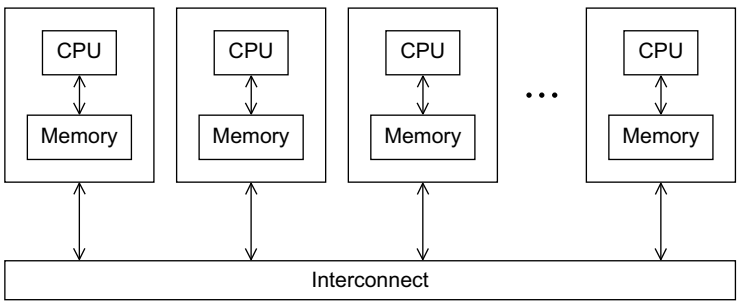


FIGURE 3.1

A distributed-memory system

An Introduction to Parallel Programming. DOI: 10.1016/B978-0-12-374260-5.00003-8

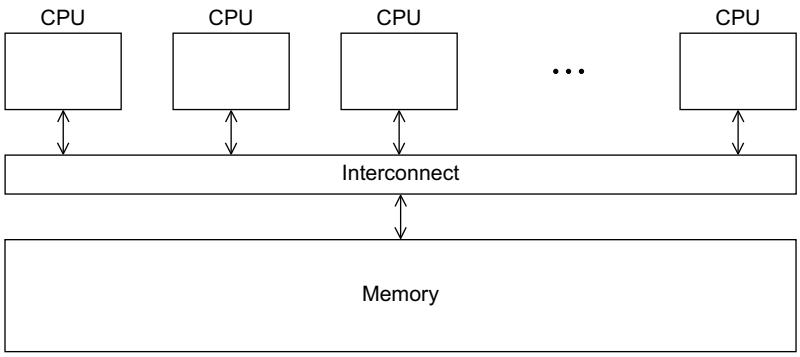


FIGURE 3.2

A shared-memory system

fundamental issues involved in writing message-passing programs–issues such as data partitioning and I/O in distributed-memory systems. We'll also revisit the issue of parallel program performance.

3.1 GETTING STARTED

Perhaps the first program that many of us saw was some variant of the "hello, world" program in Kernighan and Ritchie's classic text [29]:

```
#include <stdio.h>

int main(void) {
   printf("hello, world\n");

   return 0;
}
```

Let's write a program similar to "hello, world" that makes some use of MPI. Instead of having each process simply print a message, we'll designate one process to do the output, and the other processes will send it messages, which it will print.

In parallel programming, it's common (one might say standard) for the processes to be identified by nonnegative integer *ranks.* So if there are p processes, the processes will have ranks $0, 1, 2, \ldots, p-1$. For our parallel "hello, world," let's make process 0 the designated process, and the other processes will send it messages. See Program 3.1.

3.1.1 Compilation and execution

The details of compiling and running the program depend on your system, so you may need to check with a local expert. However, recall that when we need to be explicit, we'll assume that we're using a text editor to write the program source, and

```
1  #include <stdio.h>
2  #include <string.h>  /* For strlen             */
3  #include <mpi.h>     /* For MPI functions, etc */
4
5  const int MAX_STRING = 100;
6
7  int main(void) {
8     char       greeting[MAX_STRING];
9     int        comm_sz;  /* Number of processes */
10    int        my_rank;  /* My process rank     */
11
12    MPI_Init(NULL, NULL);
13    MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
14    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
15
16    if (my_rank != 0) {
17       sprintf(greeting, "Greetings from process %d of %d!",
18             my_rank, comm_sz);
19       MPI_Send(greeting, strlen(greeting)+1, MPI_CHAR, 0, 0,
20             MPI_COMM_WORLD);
21    } else {
22       printf("Greetings from process %d of %d!\n", my_rank,
             comm_sz);
23       for (int q = 1; q < comm_sz; q++) {
24          MPI_Recv(greeting, MAX_STRING, MPI_CHAR, q,
25             0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
26          printf("%s\n", greeting);
27       }
28    }
29
30    MPI_Finalize();
31    return 0;
32 }  /* main */
```

Program 3.1: MPI program that prints greetings from the processes

the command line to compile and run. Many systems use a command called `mpicc` for compilation:[1]

```
$ mpicc -g -Wall -o mpi_hello mpi_hello.c
```

Typically, `mpicc` is a script that's a **wrapper** for the C compiler. A **wrapper script** is a script whose main purpose is to run some program. In this case, the program is the C compiler. However, the wrapper simplifies the running of the compiler by telling it where to find the necessary header files and which libraries to link with the object file.

[1]Recall that the dollar sign (`$`) is the shell prompt, so it shouldn't be typed in. Also recall that, for the sake of explicitness, we assume that we're using the Gnu C compiler, `gcc`, and we always use the options `-g`, `-Wall`, and `-o`. See Section 2.9 for further information.

Many systems also support program startup with `mpiexec`:

```
$ mpiexec –n <number of processes> ./mpi_hello
```

So to run the program with one process, we'd type

```
$ mpiexec –n 1 ./mpi_hello
```

and to run the program with four processes, we'd type

```
$ mpiexec –n 4 ./mpi_hello
```

With one process the program's output would be

```
Greetings from process 0 of 1!
```

and with four processes the program's output would be

```
Greetings from process 0 of 4!
Greetings from process 1 of 4!
Greetings from process 2 of 4!
Greetings from process 3 of 4!
```

How do we get from invoking `mpiexec` to one or more lines of greetings? The `mpiexec` command tells the system to start `<number of processes>` instances of our `<mpi_hello>` program. It may also tell the system which core should run each instance of the program. After the processes are running, the MPI implementation takes care of making sure that the processes can communicate with each other.

3.1.2 MPI programs

Let's take a closer look at the program. The first thing to observe is that this *is* a C program. For example, it includes the standard C header files `stdio.h` and `string.h`. It also has a `main` function just like any other C program. However, there are many parts of the program which are new. Line 3 includes the `mpi.h` header file. This contains prototypes of MPI functions, macro definitions, type definitions, and so on; it contains all the definitions and declarations needed for compiling an MPI program.

The second thing to observe is that all of the identifiers defined by MPI start with the string `MPI_`. The first letter following the underscore is capitalized for function names and MPI-defined types. All of the letters in MPI-defined macros and constants are capitalized, so there's no question about what is defined by MPI and what's defined by the user program.

3.1.3 `MPI_Init` and `MPI_Finalize`

In Line 12 the call to `MPI_Init` tells the MPI system to do all of the necessary setup. For example, it might allocate storage for message buffers, and it might decide which process gets which rank. As a rule of thumb, no other MPI functions should be called before the program calls `MPI_Init`. Its syntax is

```
int MPI_Init(
      int*     argc_p  /* in/out */,
      char***  argv_p  /* in/out */);
```

The arguments, `argc_p` and `argv_p`, are pointers to the arguments to `main`, `argc`, and `argv`. However, when our program doesn't use these arguments, we can just pass `NULL` for both. Like most MPI functions, `MPI_Init` returns an `int` error code, and in most cases we'll ignore these error codes.

In Line 30 the call to `MPI_Finalize` tells the MPI system that we're done using MPI, and that any resources allocated for MPI can be freed. The syntax is quite simple:

```
int MPI_Finalize(void);
```

In general, no MPI functions should be called after the call to `MPI_Finalize`.

Thus, a typical MPI program has the following basic outline:

```
. . .
#include <mpi.h>
. . .
int main(int argc, char* argv[]) {
   . . .
   /* No MPI calls before this */
   MPI_Init(&argc, &argv);
   . . .
   MPI_Finalize();
   /* No MPI calls after this */
   . . .
   return 0;
}
```

However, we've already seen that it's not necessary to pass pointers to `argc` and `argv` to `MPI_Init`. It's also not necessary that the calls to `MPI_Init` and `MPI_Finalize` be in `main`.

3.1.4 Communicators, `MPI_Comm_size` and `MPI_Comm_rank`

In MPI a **communicator** is a collection of processes that can send messages to each other. One of the purposes of `MPI_Init` is to define a communicator that consists of all of the processes started by the user when she started the program. This communicator is called `MPI_COMM_WORLD`. The function calls in Lines 13 and 14 are getting information about `MPI_COMM_WORLD`. Their syntax is

```
int MPI_Comm_size(
      MPI_Comm  comm         /* in  */,
      int*      comm_sz_p    /* out */);

int MPI_Comm_rank(
      MPI_Comm  comm         /* in  */,
      int*      my_rank_p    /* out */);
```

For both functions, the first argument is a communicator and has the special type defined by MPI for communicators, `MPI_Comm`. `MPI_Comm_size` returns in its second argument the number of processes in the communicator, and `MPI_Comm_rank` returns in its second argument the calling process' rank in the communicator. We'll often use the variable `comm_sz` for the number of processes in `MPI_COMM_WORLD`, and the variable `my_rank` for the process rank.

3.1.5 SPMD programs

Notice that we compiled a single program—we didn't compile a different program for each process—and we did this in spite of the fact that process 0 is doing something fundamentally different from the other processes: it's receiving a series of messages and printing them, while each of the other processes is creating and sending a message. This is quite common in parallel programming. In fact, *most* MPI programs are written in this way. That is, a single program is written so that different processes carry out different actions, and this is achieved by simply having the processes branch on the basis of their process rank. Recall that this approach to parallel programming is called **single program, multiple data**, or **SPMD**. The `if-else` statement in Lines 16 through 28 makes our program SPMD.

Also notice that our program will, in principle, run with any number of processes. We saw a little while ago that it can be run with one process or four processes, but if our system has sufficient resources, we could also run it with 1000 or even 100,000 processes. Although MPI doesn't require that programs have this property, it's almost always the case that we try to write programs that will run with any number of processes, because we usually don't know in advance the exact resources available to us. For example, we might have a 20-core system available today, but tomorrow we might have access to a 500-core system.

3.1.6 Communication

In Lines 17 and 18, each process, other than process 0, creates a message it will send to process 0. (The function `sprintf` is very similar to `printf`, except that instead of writing to `stdout`, it writes to a string.) Lines 19–20 actually send the message to process 0. Process 0, on the other hand, simply prints its message using `printf`, and then uses a `for` loop to receive and print the messages sent by processes $1, 2, \ldots,$ `comm_sz` $- 1$. Lines 24–25 receive the message sent by process q, for $q = 1, 2, \ldots,$ `comm_sz` $- 1$.

3.1.7 MPI_Send

The sends executed by processes $1, 2, \ldots,$ `comm_sz` $- 1$ are fairly complex, so let's take a closer look at them. Each of the sends is carried out by a call to `MPI_Send`, whose syntax is

```
int MPI_Send(
      void*         msg_buf_p     /* in */,
      int           msg_size      /* in */,
      MPI_Datatype  msg_type      /* in */,
      int           dest          /* in */,
      int           tag           /* in */,
      MPI_Comm      communicator  /* in */);
```

The first three arguments, `msg_buf_p`, `msg_size`, and `msg_type`, determine the contents of the message. The remaining arguments, `dest`, `tag`, and `communicator`, determine the destination of the message.

The first argument, `msg_buf_p`, is a pointer to the block of memory containing the contents of the message. In our program, this is just the string containing the message, `greeting`. (Remember that in C an array, such as a string, is a pointer.) The second and third arguments, `msg_size` and `msg_type`, determine the amount of data to be sent. In our program, the `msg_size` argument is the number of characters in the message plus one character for the `'\0'` character that terminates C strings. The `msg_type` argument is `MPI_CHAR`. These two arguments together tell the system that the message contains `strlen(greeting)+1` `chars`.

Since C types (`int`, `char`, and so on.) can't be passed as arguments to functions, MPI defines a special type, `MPI_Datatype`, that is used for the `msg_type` argument. MPI also defines a number of constant values for this type. The ones we'll use (and a few others) are listed in Table 3.1.

Notice that the size of the string `greeting` is not the same as the size of the message specified by the arguments `msg_size` and `msg_type`. For example, when we run the program with four processes, the length of each of the messages is 31 characters,

Table 3.1 Some Predefined MPI Datatypes

MPI datatype	C datatype
MPI_CHAR	**signed char**
MPI_SHORT	**signed short int**
MPI_INT	**signed int**
MPI_LONG	**signed long int**
MPI_LONG_LONG	**signed long long int**
MPI_UNSIGNED_CHAR	**unsigned char**
MPI_UNSIGNED_SHORT	**unsigned short int**
MPI_UNSIGNED	**unsigned int**
MPI_UNSIGNED_LONG	**unsigned long int**
MPI_FLOAT	**float**
MPI_DOUBLE	**double**
MPI_LONG_DOUBLE	**long double**
MPI_BYTE	
MPI_PACKED	

while we've allocated storage for 100 characters in `greetings`. Of course, the size of the message sent should be less than or equal to the amount of storage in the buffer—in our case the string `greeting`.

The fourth argument, `dest`, specifies the rank of the process that should receive the message. The fifth argument, `tag`, is a nonnegative `int`. It can be used to distinguish messages that are otherwise identical. For example, suppose process 1 is sending floats to process 0. Some of the floats should be printed, while others should be used in a computation. Then the first four arguments to `MPI_Send` provide no information regarding which floats should be printed and which should be used in a computation. So process 1 can use, say, a tag of 0 for the messages that should be printed and a tag of 1 for the messages that should be used in a computation.

The final argument to `MPI_Send` is a communicator. All MPI functions that involve communication have a communicator argument. One of the most important purposes of communicators is to specify communication universes; recall that a communicator is a collection of processes that can send messages to each other. Conversely, a message sent by a process using one communicator cannot be received by a process that's using a different communicator. Since MPI provides functions for creating new communicators, this feature can be used in complex programs to insure that messages aren't "accidentally received" in the wrong place.

An example will clarify this. Suppose we're studying global climate change, and we've been lucky enough to find two libraries of functions, one for modeling the Earth's atmosphere and one for modeling the Earth's oceans. Of course, both libraries use MPI. These models were built independently, so they don't communicate with each other, but they do communicate internally. It's our job to write the interface code. One problem we need to solve is to insure that the messages sent by one library won't be accidentally received by the other. We might be able to work out some scheme with tags: the atmosphere library gets tags $0, 1, \ldots, n-1$ and the ocean library gets tags $n, n+1, \ldots, n+m$. Then each library can use the given range to figure out which tag it should use for which message. However, a much simpler solution is provided by communicators: we simply pass one communicator to the atmosphere library functions and a different communicator to the ocean library functions.

3.1.8 MPI_Recv

The first six arguments to `MPI_Recv` correspond to the first six arguments of `MPI_Send`:

```
int MPI_Recv(
      void*          msg_buf_p     /* out */,
      int            buf_size      /* in  */,
      MPI_Datatype   buf_type      /* in  */,
      int            source        /* in  */,
      int            tag           /* in  */,
      MPI_Comm       communicator  /* in  */,
      MPI_Status*    status_p      /* out */);
```

Thus, the first three arguments specify the memory available for receiving the message: `msg_buf_p` points to the block of memory, `buf_size` determines the

number of objects that can be stored in the block, and buf_type indicates the type of the objects. The next three arguments identify the message. The source argument specifies the process from which the message should be received. The tag argument should match the tag argument of the message being sent, and the communicator argument must match the communicator used by the sending process. We'll talk about the status_p argument shortly. In many cases it won't be used by the calling function, and, as in our "greetings" program, the special MPI constant MPI_STATUS_IGNORE can be passed.

3.1.9 Message matching

Suppose process *q* calls MPI_Send with

```
MPI_Send(send_buf_p, send_buf_sz, send_type, dest, send_tag,
         send_comm);
```

Also suppose that process *r* calls MPI_Recv with

```
MPI_Recv(recv_buf_p, recv_buf_sz, recv_type, src, recv_tag,
         recv_comm, &status);
```

Then the message sent by *q* with the above call to MPI_Send can be received by *r* with the call to MPI_Recv if

- recv_comm = send_comm,
- recv_tag = send_tag,
- dest = r, and
- src = q.

These conditions aren't quite enough for the message to be *successfully* received, however. The parameters specified by the first three pairs of arguments, send_buf_p/recv_buf_p, send_buf_sz/recv_buf_sz, and send_type/recv_type, must specify compatible buffers. For detailed rules, see the MPI-1 specification [39]. Most of the time, the following rule will suffice:

- If recv_type = send_type and recv_buf_sz $\geq$ send_buf_sz, then the message sent by *q* can be successfully received by *r*.

Of course, it can happen that one process is receiving messages from multiple processes, *and* the receiving process doesn't know the order in which the other processes will send the messages. For example, suppose, for example, process 0 is doling out work to processes $1, 2, \ldots,$ comm_sz $- 1$, and processes $1, 2, \ldots,$ comm_sz $- 1$, send their results back to process 0 when they finish the work. If the work assigned to each process takes an unpredictable amount of time, then 0 has no way of knowing the order in which the processes will finish. If process 0 simply receives the results in process rank order—first the results from process 1, then the results from process 2, and so on—and if, say, process comm_sz-1 finishes first, it *could* happen that process comm_sz-1 could sit and wait for the other processes to finish. In order to avoid this problem, MPI provides a special constant MPI_ANY_SOURCE that can be passed to

MPI_Recv. Then, if process 0 executes the following code, it can receive the results in the order in which the processes finish:

```
for (i = 1; i < comm_sz; i++) {
   MPI_Recv(result, result_sz, result_type, MPI_ANY_SOURCE,
         result_tag, comm, MPI_STATUS_IGNORE);
   Process_result(result);
}
```

Similarly, it's possible that one process can be receiving multiple messages with different tags from another process, and the receiving process doesn't know the order in which the messages will be sent. For this circumstance, MPI provides the special constant MPI_ANY_TAG that can be passed to the tag argument of MPI_Recv.

A couple of points should be stressed in connection with these "wildcard" arguments:

1. Only a receiver can use a wildcard argument. Senders must specify a process rank and a nonnegative tag. Thus, MPI uses a "push" communication mechanism rather than a "pull" mechanism.
2. There is no wildcard for communicator arguments; both senders and receivers must always specify communicators.

3.1.10 The status_p argument

If you think about these rules for a minute, you'll notice that a receiver can receive a message without knowing

1. the amount of data in the message,
2. the sender of the message, or
3. the tag of the message.

So how can the receiver find out these values? Recall that the last argument to MPI_Recv has type MPI_Status*. The MPI type MPI_Status is a struct with at least the three members MPI_SOURCE, MPI_TAG, and MPI_ERROR. Suppose our program contains the definition

```
MPI_Status status;
```

Then, after a call to MPI_Recv in which &status is passed as the last argument, we can determine the sender and tag by examining the two members

```
status.MPI_SOURCE
status.MPI_TAG
```

The amount of data that's been received isn't stored in a field that's directly accessible to the application program. However, it can be retrieved with a call to MPI_Get_count. For example, suppose that in our call to MPI_Recv, the type of the receive buffer is recv_type and, once again, we passed in &status. Then the call

```
MPI_Get_count(&status, recv_type, &count)
```

will return the number of elements received in the `count` argument. In general, the syntax of `MPI_Get_count` is

```
int MPI_Get_count(
      MPI_Status*  status_p  /* in  */,
      MPI_Datatype type      /* in  */,
      int*         count_p   /* out */);
```

Note that the count isn't directly accessible as a member of the `MPI_Status` variable simply because it depends on the type of the received data, and, consequently, determining it would probably require a calculation (e.g. (number of bytes received)/(bytes per object)). If this information isn't needed, we shouldn't waste a calculation determining it.

3.1.11 Semantics of `MPI_Send` and `MPI_Recv`

What exactly happens when we send a message from one process to another? Many of the details depend on the particular system, but we can make a few generalizations. The sending process will assemble the message. For example, it will add the "envelope" information to the actual data being transmitted—the destination process rank, the sending process rank, the tag, the communicator, and some information on the size of the message. Once the message has been assembled, recall from Chapter 2 that there are essentially two possibilities: the sending process can **buffer** the message or it can **block**. If it buffers the message, the MPI system will place the message (data and envelope) into its own internal storage, and the call to `MPI_Send` will return.

Alternatively, if the system blocks, it will wait until it can begin transmitting the message, and the call to `MPI_Send` may not return immediately. Thus, if we use `MPI_Send`, when the function returns, we don't actually know whether the message has been transmitted. We only know that the storage we used for the message, the send buffer, is available for reuse by our program. If we need to know that the message has been transmitted, or if we need for our call to `MPI_Send` to return immediately—regardless of whether the message has been sent—MPI provides alternative functions for sending. We'll learn about one of these alternative functions later.

The exact behavior of `MPI_Send` is determined by the MPI implementation. However, typical implementations have a default "cutoff" message size. If the size of a message is less than the cutoff, it will be buffered. If the size of the message is greater than the cutoff, `MPI_Send` will block.

Unlike `MPI_Send`, `MPI_Recv` always blocks until a matching message has been received. Thus, when a call to `MPI_Recv` returns, we know that there is a message stored in the receive buffer (unless there's been an error). There is an alternate method for receiving a message, in which the system checks whether a matching message is available and returns, regardless of whether there is one. (For more details on the use of nonblocking communication, see Exercise 6.22.)

MPI requires that messages be **nonovertaking**. This means that if process q sends two messages to process r, then the first message sent by q must be available to r

before the second message. However, there is no restriction on the arrival of messages sent from different processes. That is, if q and t both send messages to r, then even if q sends its message before t sends its message, there is no requirement that q's message become available to r before t's message. This is essentially because MPI can't impose performance on a network. For example, if q happens to be running on a machine on Mars, while r and t are both running on the same machine in San Francisco, and if q sends its message a nanosecond before t sends its message, it would be extremely unreasonable to require that q's message arrive before t's.

3.1.12 Some potential pitfalls

Note that the semantics of `MPI_Recv` suggests a potential pitfall in MPI programming: If a process tries to receive a message and there's no matching send, then the process will block forever. That is, the process will **hang**. When we design our programs, we therefore need to be sure that every receive has a matching send. Perhaps even more important, we need to be very careful when we're coding that there are no inadvertent mistakes in our calls to `MPI_Send` and `MPI_Recv`. For example, if the tags don't match, or if the rank of the destination process is the same as the rank of the source process, the receive won't match the send, and either a process will hang, or, perhaps worse, the receive may match *another* send.

Similarly, if a call to `MPI_Send` blocks and there's no matching receive, then the sending process can hang. If, on the other hand, a call to `MPI_Send` is buffered and there's no matching receive, then the message will be lost.

3.2 THE TRAPEZOIDAL RULE IN MPI

Printing messages from processes is all well and good, but we're probably not taking the trouble to learn to write MPI programs just to print messages. Let's take a look at a somewhat more useful program—let's write a program that implements the trapezoidal rule for numerical integration.

3.2.1 The trapezoidal rule

Recall that we can use the trapezoidal rule to approximate the area between the graph of a function, $y = f(x)$, two vertical lines, and the x-axis. See Figure 3.3. The basic idea is to divide the interval on the x-axis into n equal subintervals. Then we approximate the area lying between the graph and each subinterval by a trapezoid whose base is the subinterval, whose vertical sides are the vertical lines through the endpoints of the subinterval, and whose fourth side is the secant line joining the points where the vertical lines cross the graph. See Figure 3.4. If the endpoints of the subinterval are x_i and x_{i+1}, then the length of the subinterval is $h = x_{i+1} - x_i$. Also, if the lengths of the two vertical segments are $f(x_i)$ and $f(x_{i+1})$, then the area of the trapezoid is

$$\text{Area of one trapezoid} = \frac{h}{2}[f(x_i) + f(x_{i+1})].$$

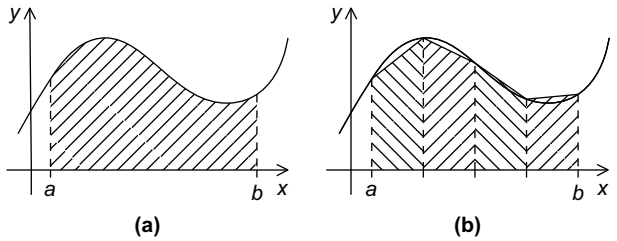


FIGURE 3.3

The trapezoidal rule: (a) area to be estimated and (b) approximate area using trapezoids

Since we chose the n subintervals so that they would all have the same length, we also know that if the vertical lines bounding the region are $x = a$ and $x = b$, then

$$h = \frac{b-a}{n}.$$

Thus, if we call the leftmost endpoint x_0, and the rightmost endpoint x_n, we have

$$x_0 = a,\ x_1 = a+h,\ x_2 = a+2h, \ldots,\ x_{n-1} = a+(n-1)h,\ x_n = b,$$

and the sum of the areas of the trapezoids—our approximation to the total area—is

$$\text{Sum of trapezoid areas } = h[f(x_0)/2 + f(x_1) + f(x_2) + \cdots + f(x_{n-1}) + f(x_n)/2].$$

Thus, pseudo-code for a serial program might look something like this:

```
/* Input:  a, b, n */
h = (b-a)/n;
approx = (f(a) + f(b))/2.0;
for (i = 1; i <= n-1; i++) {
   x_i = a + i*h;
   approx += f(x_i);
}
approx = h*approx;
```

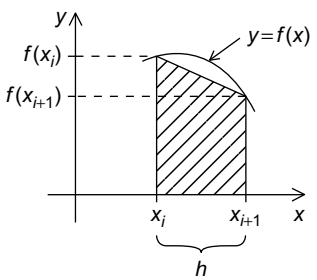


FIGURE 3.4

One trapezoid

3.2.2 Parallelizing the trapezoidal rule

It is not the most attractive word, but, as we noted in Chapter 1, people who write parallel programs do use the verb "parallelize" to describe the process of converting a serial program or algorithm into a parallel program.

Recall that we can design a parallel program using four basic steps:

1. Partition the problem solution into tasks.
2. Identify the communication channels between the tasks.
3. Aggregate the tasks into composite tasks.
4. Map the composite tasks to cores.

In the partitioning phase, we usually try to identify as many tasks as possible. For the trapezoidal rule, we might identify two types of tasks: one type is finding the area of a single trapezoid, and the other is computing the sum of these areas. Then the communication channels will join each of the tasks of the first type to the single task of the second type. See Figure 3.5.

So how can we aggregate the tasks and map them to the cores? Our intuition tells us that the more trapezoids we use, the more accurate our estimate will be. That is, we should use many trapezoids, and we will use many more trapezoids than cores. Thus, we need to aggregate the computation of the areas of the trapezoids into groups. A natural way to do this is to split the interval $[a,b]$ up into `comm_sz` subintervals. If `comm_sz` evenly divides n, the number of trapezoids, we can simply apply the trapezoidal rule with $n/$`comm_sz` trapezoids to each of the `comm_sz` subintervals. To finish, we can have one of the processes, say process 0, add the estimates.

Let's make the simplifying assumption that `comm_sz` evenly divides n. Then pseudo-code for the program might look something like the following:

```
1  Get a, b, n;
2  h = (b-a)/n;
3  local_n = n/comm_sz;
4  local_a = a + my_rank*local_n*h;
5  local_b = local_a + local_n*h;
6  local_integral = Trap(local_a, local_b, local_n, h);
```

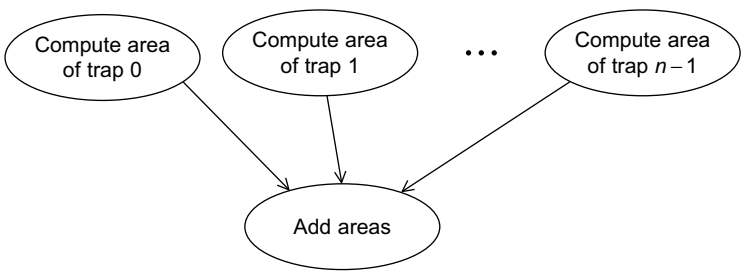


FIGURE 3.5

Tasks and communications for the trapezoidal rule

```
 7  if (my_rank != 0)
 8     Send local_integral to process 0;
 9  else /* my_rank == 0 */
10     total_integral = local_integral;
11     for (proc = 1; proc < comm_sz; proc++) {
12        Receive local_integral from proc;
13        total_integral += local_integral;
14     }
15  }
16  if (my_rank == 0)
17     print result;
```

Let's defer, for the moment, the issue of input and just "hardwire" the values for a, b, and n. When we do this, we get the MPI program shown in Program 3.2. The `Trap` function is just an implementation of the serial trapezoidal rule. See Program 3.3.

Notice that in our choice of identifiers, we try to differentiate between *local* and *global* variables. **Local** variables are variables whose contents are significant only on the process that's using them. Some examples from the trapezoidal rule program are `local_a`, `local_b`, and `local_n`. Variables whose contents are significant to all the processes are sometimes called **global** variables. Some examples from the trapezoidal rule are `a`, `b`, and `n`. Note that this usage is different from the usage you learned in your introductory programming class, where local variables are private to a single function and global variables are accessible to all the functions. However, no confusion should arise, since the context will usually make the meaning clear.

3.3 DEALING WITH I/O

Of course, the current version of the parallel trapezoidal rule has a serious deficiency: it will only compute the integral over the interval $[0, 3]$ using 1024 trapezoids. We can edit the code and recompile, but this is quite a bit of work compared to simply typing in three new numbers. We need to address the problem of getting input from the user. While we're talking about input to parallel programs, it might be a good idea to also take a look at output. We discussed these two issues in Chapter 2, so if you remember the discussion of nondeterminism and output, you can skip ahead to Section 3.3.2.

3.3.1 Output

In both the "greetings" program and the trapezoidal rule program we've assumed that process 0 can write to `stdout`, that is, its calls to `printf` behave as we might expect. Although the MPI standard doesn't specify which processes have access to which I/O devices, virtually all MPI implementations allow *all* the processes in `MPI_COMM_WORLD` full access to `stdout` and `stderr`, so most MPI implementations allow all processes to execute `printf` and `fprintf(stderr, ...)`.

However, most MPI implementations don't provide any automatic scheduling of access to these devices. That is, if multiple processes are attempting to write to,

```
1  int main(void) {
2     int my_rank, comm_sz, n = 1024, local_n;
3     double a = 0.0, b = 3.0, h, local_a, local_b;
4     double local_int, total_int;
5     int source;
6
7     MPI_Init(NULL, NULL);
8     MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
9     MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
10
11    h = (b-a)/n;          /* h is the same for all processes */
12    local_n = n/comm_sz;  /* So is the number of trapezoids  */
13
14    local_a = a + my_rank*local_n*h;
15    local_b = local_a + local_n*h;
16    local_int = Trap(local_a, local_b, local_n, h);
17
18    if (my_rank != 0) {
19       MPI_Send(&local_int, 1, MPI_DOUBLE, 0, 0,
20             MPI_COMM_WORLD);
21    } else {
22       total_int = local_int;
23       for (source = 1; source < comm_sz; source++) {
24          MPI_Recv(&local_int, 1, MPI_DOUBLE, source, 0,
25                MPI_COMM_WORLD, MPI_STATUS_IGNORE);
26          total_int += local_int;
27       }
28    }
29
30    if (my_rank == 0) {
31       printf("With n = %d trapezoids, our estimate\n", n);
32       printf("of the integral from %f to %f = %.15e\n",
33           a, b, total_int);
34    }
35    MPI_Finalize();
36    return 0;
37 } /*  main  */
```

Program 3.2: First version of the MPI trapezoidal rule

say, `stdout`, the order in which the processes' output appears will be unpredictable. Indeed, it can even happen that the output of one process will be interrupted by the output of another process.

For example, suppose we try to run an MPI program in which each process simply prints a message. See Program 3.4. On our cluster, if we run the program with five processes, it often produces the "expected" output:

```
Proc 0 of 5 > Does anyone have a toothpick?
Proc 1 of 5 > Does anyone have a toothpick?
Proc 2 of 5 > Does anyone have a toothpick?
```

```
1  double Trap(
2        double left_endpt  /* in */,
3        double right_endpt /* in */,
4        int    trap_count  /* in */,
5        double base_len    /* in */) {
6     double estimate, x;
7     int i;
8
9     estimate = (f(left_endpt) + f(right_endpt))/2.0;
10    for (i = 1; i <= trap_count-1; i++) {
11       x = left_endpt + i*base_len;
12       estimate += f(x);
13    }
14    estimate = estimate*base_len;
15
16    return estimate;
17 } /*  Trap  */
```

Program 3.3: `Trap` function in the MPI trapezoidal rule

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

int main(void) {
   int my_rank, comm_sz;

   MPI_Init(NULL, NULL);
   MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

   printf("Proc %d of %d > Does anyone have a toothpick?\n",
         my_rank, comm_sz);

   MPI_Finalize();
   return 0;
}  /* main */
```

Program 3.4: Each process just prints a message

```
Proc 3 of 5 > Does anyone have a toothpick?
Proc 4 of 5 > Does anyone have a toothpick?
```

However, when we run it with six processes, the order of the output lines is unpredictable:

```
Proc 0 of 6 > Does anyone have a toothpick?
Proc 1 of 6 > Does anyone have a toothpick?
Proc 2 of 6 > Does anyone have a toothpick?
```

```
Proc 5 of 6 > Does anyone have a toothpick?
Proc 3 of 6 > Does anyone have a toothpick?
Proc 4 of 6 > Does anyone have a toothpick?
```

or

```
Proc 0 of 6 > Does anyone have a toothpick?
Proc 1 of 6 > Does anyone have a toothpick?
Proc 2 of 6 > Does anyone have a toothpick?
Proc 4 of 6 > Does anyone have a toothpick?
Proc 3 of 6 > Does anyone have a toothpick?
Proc 5 of 6 > Does anyone have a toothpick?
```

The reason this happens is that the MPI processes are "competing" for access to the shared output device, `stdout`, and it's impossible to predict the order in which the processes' output will be queued up. Such a competition results in **nondeterminism**. That is, the actual output will vary from one run to the next.

In any case, if we don't want output from different processes to appear in a random order, it's up to us to modify our program accordingly. For example, we can have each process other than 0 send its output to process 0, and process 0 can print the output in process rank order. This is exactly what we did in the "greetings" program.

3.3.2 Input

Unlike output, most MPI implementations only allow process 0 in `MPI_COMM_WORLD` access to `stdin`. This makes sense: If multiple processes have access to `stdin`, which process should get which parts of the input data? Should process 0 get the first line? Process 1 the second? Or should process 0 get the first character?

In order to write MPI programs that can use `scanf`, we need to branch on process rank, with process 0 reading in the data and then sending it to the other processes. For example, we might write the `Get_input` function shown in Program 3.5 for our parallel trapezoidal rule program. In this function, process 0 simply reads in the values for a, b, and n and sends all three values to each process. This function uses the same basic communication structure as the "greetings" program, except that now process 0 is sending to each process, while the other processes are receiving.

To use this function, we can simply insert a call to it inside our main function, being careful to put it after we've initialized `my_rank` and `comm_sz`:

```
. . .
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);

Get_data(my_rank, comm_sz, &a, &b, &n);

h = (b-a)/n;
. . .
```

```
1  void Get_input(
2        int      my_rank   /* in  */,
3        int      comm_sz   /* in  */,
4        double*  a_p       /* out */,
5        double*  b_p       /* out */,
6        int*     n_p       /* out */) {
7     int dest;
8
9     if (my_rank == 0) {
10       printf("Enter a, b, and n\n");
11       scanf("%lf %lf %d", a_p, b_p, n_p);
12       for (dest = 1; dest < comm_sz; dest++) {
13          MPI_Send(a_p, 1, MPI_DOUBLE, dest, 0, MPI_COMM_WORLD);
14          MPI_Send(b_p, 1, MPI_DOUBLE, dest, 0, MPI_COMM_WORLD);
15          MPI_Send(n_p, 1, MPI_INT, dest, 0, MPI_COMM_WORLD);
16       }
17    } else { /* my_rank != 0 */
18       MPI_Recv(a_p, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD,
19             MPI_STATUS_IGNORE);
20       MPI_Recv(b_p, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD,
21             MPI_STATUS_IGNORE);
22       MPI_Recv(n_p, 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
23             MPI_STATUS_IGNORE);
24    }
25 }  /* Get_input */
```

Program 3.5: A function for reading user input

3.4 COLLECTIVE COMMUNICATION

If we pause for a moment and think about our trapezoidal rule program, we can find several things that we might be able to improve on. One of the most obvious is that the "global sum" after each process has computed its part of the integral. If we hire eight workers to, say, build a house, we might feel that we weren't getting our money's worth if seven of the workers told the first what to do, and then the seven collected their pay and went home. But this is very similar to what we're doing in our global sum. Each process with rank greater than 0 is "telling process 0 what to do" and then quitting. That is, each process with rank greater than 0 is, in effect, saying "add this number into the total." Process 0 is doing nearly all the work in computing the global sum, while the other processes are doing almost nothing. Sometimes it does happen that this is the best we can do in a parallel program, but if we imagine that we have eight students, each of whom has a number, and we want to find the sum of all eight numbers, we can certainly come up with a more equitable distribution of the work than having seven of the eight give their numbers to one of the students and having the first do the addition.

3.4.1 Tree-structured communication

As we already saw in Chapter 1 we might use a "binary tree structure" like that illustrated in Figure 3.6. In this diagram, initially students or processes 1, 3, 5, and 7 send their values to processes 0, 2, 4, and 6, respectively. Then processes 0, 2, 4, and 6 add the received values to their original values, and the process is repeated twice:

1. **a.** Processes 2 and 6 send their new values to processes 0 and 4, respectively.
   b. Processes 0 and 4 add the received values into their new values.
2. **a.** Process 4 sends its newest value to process 0.
   b. Process 0 adds the received value to its newest value.

This solution may not seem ideal, since half the processes (1, 3, 5, and 7) are doing the same amount of work that they did in the original scheme. However, if you think about it, the original scheme required `comm_sz` $- 1 =$ seven receives and seven adds by process 0, while the new scheme only requires three, and all the other processes do no more than two receives and adds. Furthermore, the new scheme has a property by which a lot of the work is done concurrently by different processes. For example, in the first phase, the receives and adds by processes 0, 2, 4, and 6 can all take place simultaneously. So, if the processes start at roughly the same time, the total time required to compute the global sum will be the time required by process 0, that is, three receives and three additions. We've thus reduced the overall time by more than 50%. Furthermore, if we use more processes, we can do even better. For example, if `comm_sz` $= 1024$, then the original scheme requires process 0 to do 1023 receives and additions, while it can be shown (Exercise 3.5) that the new scheme requires process 0 to do only 10 receives and additions. This improves the original scheme by more than a factor of 100!

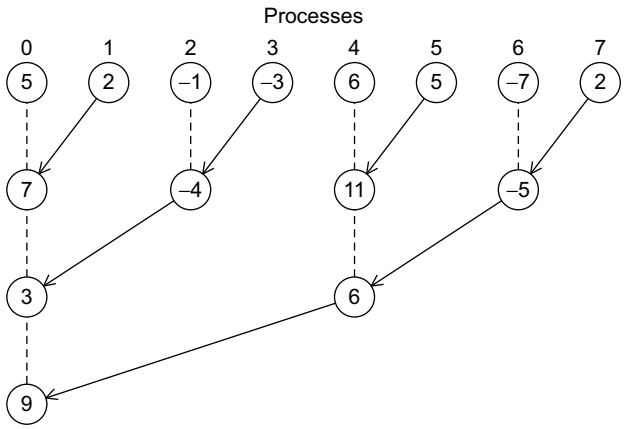


FIGURE 3.6

A tree-structured global sum

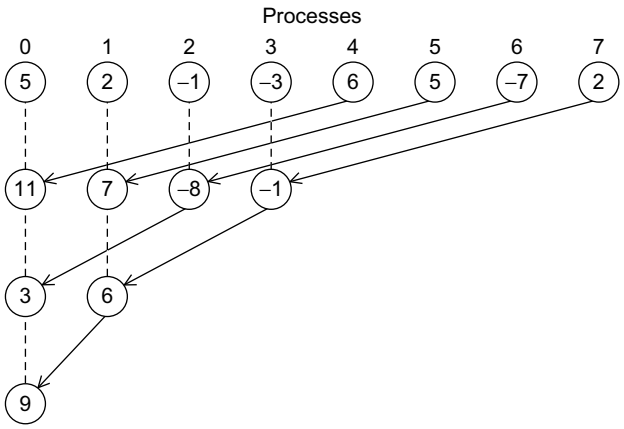


FIGURE 3.7

An alternative tree-structured global sum

You may be thinking to yourself, this is all well and good, but coding this tree-structured global sum looks like it would take a quite a bit of work, and you'd be right. See Programming Assignment 3.3. In fact, the problem may be even harder. For example, it's perfectly feasible to construct a tree-structured global sum that uses different "process-pairings." For example, we might pair 0 and 4, 1 and 5, 2 and 6, and 3 and 7 in the first phase. Then we could pair 0 and 2, and 1 and 3 in the second, and 0 and 1 in the final. See Figure 3.7. Of course, there are many other possibilities. How can we decide which is the best? Do we need to code each alternative and evaluate its performance? If we do, is it possible that one method works best for "small" trees, while another works best for "large" trees? Even worse, one approach might work best on system A, while another might work best on system B.

3.4.2 MPI_Reduce

With virtually limitless possibilities, it's unreasonable to expect each MPI programmer to write an optimal global-sum function, so MPI specifically protects programmers against this trap of endless optimization by requiring that MPI implementations include implementations of global sums. This places the burden of optimization on the developer of the MPI implementation, rather than the application developer. The assumption here is that the developer of the MPI implementation should know enough about both the hardware and the system software so that she can make better decisions about implementation details.

Now, a "global-sum function" will obviously require communication. However, unlike the `MPI_Send`-`MPI_Recv` pair, the global-sum function may involve more than two processes. In fact, in our trapezoidal rule program it will involve all the processes in `MPI_COMM_WORLD`. In MPI parlance, communication functions that involve all the processes in a communicator are called **collective communications**. To distinguish

between collective communications and functions such as MPI_Send and MPI_Recv, MPI_Send and MPI_Recv are often called **point-to-point** communications.

In fact, global sum is just a special case of an entire class of collective communications. For example, it might happen that instead of finding the sum of a collection of comm_sz numbers distributed among the processes, we want to find the maximum or the minimum or the product or any one of many other possibilities. MPI generalized the global-sum function so that any one of these possibilities can be implemented with a single function:

```
int MPI_Reduce(
      void*         input_data_p   /* in  */,
      void*         output_data_p  /* out */,
      int           count          /* in  */,
      MPI_Datatype  datatype       /* in  */,
      MPI_Op        operator       /* in  */,
      int           dest_process   /* in  */,
      MPI_Comm      comm           /* in  */);
```

The key to the generalization is the fifth argument, operator. It has type MPI_Op, which is a predefined MPI type like MPI_Datatype and MPI_Comm. There are a number of predefined values in this type. See Table 3.2. It's also possible to define your own operators; for details, see the MPI-1 Standard [39].

The operator we want is MPI_SUM. Using this value for the operator argument, we can replace the code in Lines 18 through 28 of Program 3.2 with the single function call

```
MPI_Reduce(&local_int, &total_int, 1, MPI_DOUBLE, MPI_SUM, 0,
      MPI_COMM_WORLD);
```

One point worth noting is that by using a count argument greater than 1, MPI_Reduce can operate on arrays instead of scalars. The following code could thus be used to

Table 3.2 Predefined Reduction Operators in MPI

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

add a collection of N-dimensional vectors, one per process:

```
double local_x[N], sum[N];
. . .
MPI_Reduce(local_x, sum, N, MPI_DOUBLE, MPI_SUM, 0,
      MPI_COMM_WORLD);
```

3.4.3 Collective vs. point-to-point communications

It's important to remember that collective communications differ in several ways from point-to-point communications:

1. *All* the processes in the communicator must call the same collective function. For example, a program that attempts to match a call to `MPI_Reduce` on one process with a call to `MPI_Recv` on another process is erroneous, and, in all likelihood, the program will hang or crash.
2. The arguments passed by each process to an MPI collective communication must be "compatible." For example, if one process passes in 0 as the `dest_process` and another passes in 1, then the outcome of a call to `MPI_Reduce` is erroneous, and, once again, the program is likely to hang or crash.
3. The `output_data_p` argument is only used on `dest_process`. However, all of the processes still need to pass in an actual argument corresponding to `output_data_p`, even if it's just `NULL`.
4. Point-to-point communications are matched on the basis of tags and communicators. Collective communications don't use tags, so they're matched solely on the basis of the communicator and the *order* in which they're called. As an example, consider the calls to `MPI_Reduce` shown in Table 3.3. Suppose that each process calls `MPI_Reduce` with operator `MPI_SUM`, and destination process 0. At first glance, it might seem that after the two calls to `MPI_Reduce`, the value of `b` will be three, and the value of `d` will be six. However, the names of the memory locations are irrelevant to the matching, of the calls to `MPI_Reduce`. The *order* of the calls will determine the matching, so the value stored in `b` will be $1 + 2 + 1 = 4$, and the value stored in `d` will be $2 + 1 + 2 = 5$.

A final caveat: it might be tempting to call `MPI_Reduce` using the same buffer for both input and output. For example, if we wanted to form the global sum of `x` on each process and store the result in `x` on process 0, we might try calling

```
MPI_Reduce(&x, &x, 1, MPI_DOUBLE, MPI_SUM, 0, comm);
```

Table 3.3 Multiple Calls to `MPI_Reduce`

Time	Process 0	Process 1	Process 2
0	`a = 1; c = 2`	`a = 1; c = 2`	`a = 1; c = 2`
1	`MPI_Reduce(&a, &b, ...)`	`MPI_Reduce(&c, &d, ...)`	`MPI_Reduce(&a, &b, ...)`
2	`MPI_Reduce(&c, &d, ...)`	`MPI_Reduce(&a, &b, ...)`	`MPI_Reduce(&c, &d, ...)`

However, this call is illegal in MPI, so its result will be unpredictable: it might produce an incorrect result, it might cause the program to crash, it might even produce a correct result. It's illegal because it involves **aliasing** of an output argument. Two arguments are aliased if they refer to the same block of memory, and MPI prohibits aliasing of arguments if one of them is an output or input/output argument. This is because the MPI Forum wanted to make the Fortran and C versions of MPI as similar as possible, and Fortran prohibits aliasing. In some instances, MPI provides an alternative construction that effectively avoids this restriction. See Section 6.1.9 for an example.

3.4.4 MPI_Allreduce

In our trapezoidal rule program, we just print the result, so it's perfectly natural for only one process to get the result of the global sum. However, it's not difficult to imagine a situation in which *all* of the processes need the result of a global sum in order to complete some larger computation. In this situation, we encounter some of the same problems we encountered with our original global sum. For example, if we use a tree to compute a global sum, we might "reverse" the branches to distribute the global sum (see Figure 3.8). Alternatively, we might have the processes *exchange* partial results instead of using one-way communications. Such a communication pattern is sometimes called a **butterfly** (see Figure 3.9). Once again, we don't want to have to decide on which structure to use, or how to code it for optimal performance. Fortunately, MPI provides a variant of `MPI_Reduce` that will store the result on all the processes in the communicator:

```
int MPI_Allreduce(
          void*        input_data_p   /* in  */,
          void*        output_data_p  /* out */,
          int          count          /* in  */,
          MPI_Datatype datatype       /* in  */,
          MPI_Op       operator       /* in  */,
          MPI_Comm     comm           /* in  */);
```

The argument list is identical to that for `MPI_Reduce` except that there is no `dest_process` since all the processes should get the result.

3.4.5 Broadcast

If we can improve the performance of the global sum in our trapezoidal rule program by replacing a loop of receives on process 0 with a tree-structured communication, we ought to be able to do something similar with the distribution of the input data. In fact, if we simply "reverse" the communications in the tree-structured global sum in Figure 3.6, we obtain the tree-structured communication shown in Figure 3.10, and we can use this structure to distribute the input data. A collective communication in which data belonging to a single process is sent to all of the processes in the communicator is called a **broadcast**, and you've probably guessed that MPI provides

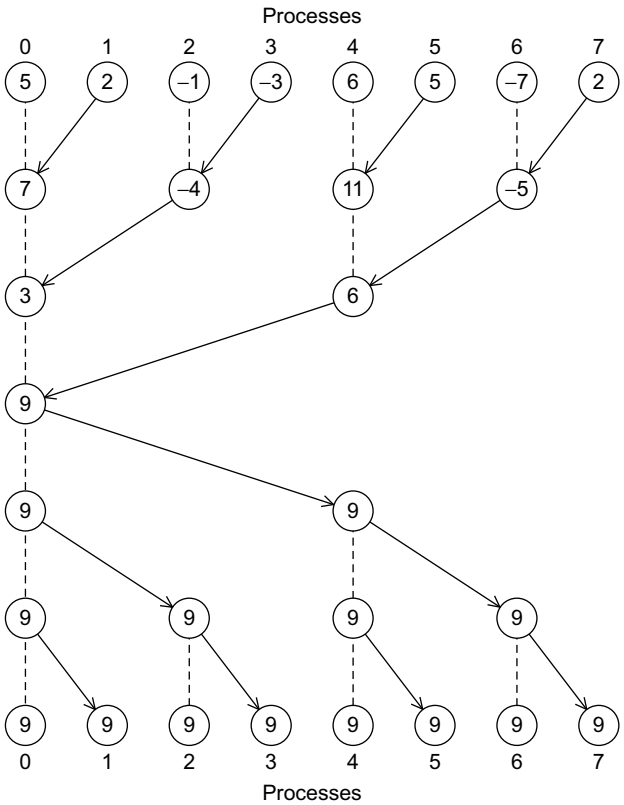


FIGURE 3.8

A global sum followed by distribution of the result

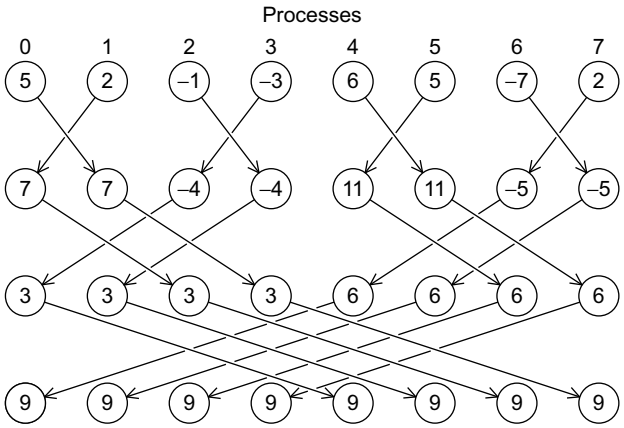


FIGURE 3.9

A butterfly-structured global sum

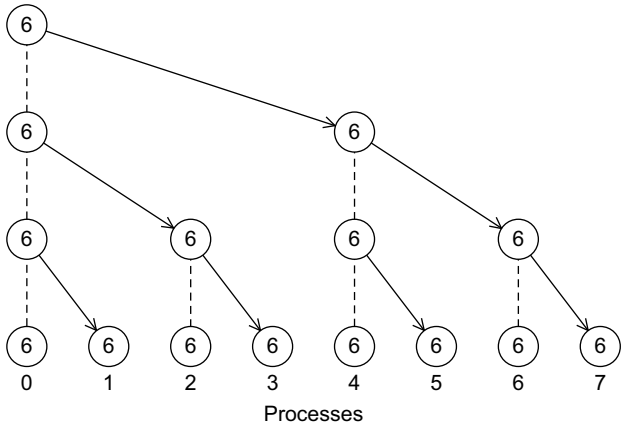


FIGURE 3.10

A tree-structured broadcast

a broadcast function:

```
int MPI_Bcast(
         void*        data_p          /* in/out */,
         int          count           /* in     */,
         MPI_Datatype datatype        /* in     */,
         int          source_proc     /* in     */,
         MPI_Comm     comm            /* in     */);
```

The process with rank `source_proc` sends the contents of the memory referenced by `data_p` to all the processes in the communicator `comm`. Program 3.6 shows how

```
1  void Get_input(
2        int      my_rank  /* in  */,
3        int      comm_sz  /* in  */,
4        double*  a_p      /* out */,
5        double*  b_p      /* out */,
6        int*     n_p      /* out */) {
7
8     if (my_rank == 0) {
9        printf("Enter a, b, and n\n");
10       scanf("%lf %lf %d", a_p, b_p, n_p);
11    }
12    MPI_Bcast(a_p, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
13    MPI_Bcast(b_p, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
14    MPI_Bcast(n_p, 1, MPI_INT, 0, MPI_COMM_WORLD);
15 }  /* Get_input */
```

Program 3.6: A version of `Get_input` that uses `MPI_Bcast`

to modify the `Get_input` function shown in Program 3.5 so that it uses `MPI_Bcast` instead of `MPI_Send` and `MPI_Recv`.

Recall that in serial programs, an in/out argument is one whose value is both used and changed by the function. For `MPI_Bcast`, however, the `data_p` argument is an input argument on the process with rank `source_proc` and an output argument on the other processes. Thus, when an argument to a collective communication is labeled in/out, it's possible that it's an input argument on some processes and an output argument on other processes.

3.4.6 Data distributions

Suppose we want to write a function that computes a vector sum:

$$\begin{aligned}\mathbf{x}+\mathbf{y} &= (x_0, x_1, \ldots, x_{n-1}) + (y_0, y_1, \ldots, y_{n-1}) \\ &= (x_0 + y_0, x_1 + y_1, \ldots, x_{n-1} + y_{n-1}) \\ &= (z_0, z_1, \ldots, z_{n-1}) \\ &= \mathbf{z}\end{aligned}$$

If we implement the vectors as arrays of, say, **double**s, we could implement serial vector addition with the code shown in Program 3.7.

```
1  void Vector_sum(double x[], double y[], double z[], int n) {
2     int i;
3
4     for (i = 0; i < n; i++)
5        z[i] = x[i] + y[i];
6  }  /* Vector_sum */
```

Program 3.7: A serial implementation of vector addition

How could we implement this using MPI? The work consists of adding the individual components of the vectors, so we might specify that the tasks are just the additions of corresponding components. Then there is no communication between the tasks, and the problem of parallelizing vector addition boils down to aggregating the tasks and assigning them to the cores. If the number of components is n and we have `comm_sz` cores or processes, let's assume that n evenly divides `comm_sz` and define `local_n` $= n/$`comm_sz`. Then we can simply assign blocks of `local_n` consecutive components to each process. The four columns on the left of Table 3.4 show an example when $n = 12$ and `comm_sz` $= 3$. This is often called a **block partition** of the vector.

An alternative to a block partition is a **cyclic partition**. In a cyclic partition, we assign the components in a round robin fashion. The four columns in the middle of Table 3.4 show an example when $n = 12$ and `comm_sz` $= 3$. Process 0 gets component 0, process 1 gets component 1, process 2 gets component 2, process 0 gets component 3, and so on.

Table 3.4 Different Partitions of a 12-Component Vector among Three Processes

	Components											
									Block-Cyclic			
Process	Block				Cyclic				Blocksize = 2			
0	0	1	2	3	0	3	6	9	0	1	6	7
1	4	5	6	7	1	4	7	10	+6'	3	8	9
2	8	9	10	11	2	5	8	11	4	5	10	11

A third alternative is a **block-cyclic partition**. The idea here is that instead of using a cyclic distribution of individual components, we use a cyclic distribution of *blocks* of components, so a block-cyclic distribution isn't fully specified until we decide how large the blocks are. If `comm_sz` $= 3$, $n = 12$, and the blocksize $b = 2$, an example is shown in the four columns on the right of Table 3.4.

Once we've decided how to partition the vectors, it's easy to write a parallel vector addition function: each process simply adds its assigned components. Furthermore, regardless of the partition, each process will have `local_n` components of the vector, and, in order to save on storage, we can just store these on each process as an array of `local_n` elements. Thus, each process will execute the function shown in Program 3.8. Although the names of the variables have been changed to emphasize the fact that the function is operating on only the process' portion of the vector, this function is virtually identical to the original serial function.

```
1  void Parallel_vector_sum(
2        double  local_x[]  /* in  */,
3        double  local_y[]  /* in  */,
4        double  local_z[]  /* out */,
5        int     local_n    /* in  */) {
6     int local_i;
7
8     for (local_i = 0; local_i < local_n; local_i++)
9        local_z[local_i] = local_x[local_i] + local_y[local_i];
10 }  /* Parallel_vector_sum */
```

Program 3.8: A parallel implementation of vector addition

3.4.7 Scatter

Now suppose we want to test our vector addition function. It would be convenient to be able to read the dimension of the vectors and then read in the vectors **x** and **y**.

We already know how to read in the dimension of the vectors: process 0 can prompt the user, read in the value, and broadcast the value to the other processes. We might try something similar with the vectors: process 0 could read them in and broadcast them to the other processes. However, this could be very wasteful. If there are 10 processes and the vectors have 10,000 components, then each process will need to allocate storage for vectors with 10,000 components, when it is only operating on subvectors with 1000 components. If, for example, we use a block distribution, it would be better if process 0 sent only components 1000 to 1999 to process 1, components 2000 to 2999 to process 2, and so on. Using this approach, processes 1 to 9 would only need to allocate storage for the components they're actually using.

Thus, we might try writing a function that reads in an entire vector that is on process 0 but only sends the needed components to each of the other processes. For the communication MPI provides just such a function:

```
int MPI_Scatter(
      void*         send_buf_p  /* in  */,
      int           send_count  /* in  */,
      MPI_Datatype  send_type   /* in  */,
      void*         recv_buf_p  /* out */,
      int           recv_count  /* in  */,
      MPI_Datatype  recv_type   /* in  */,
      int           src_proc    /* in  */,
      MPI_Comm      comm        /* in  */);
```

If the communicator `comm` contains `comm_sz` processes, then `MPI_Scatter` divides the data referenced by `send_buf_p` into `comm_sz` pieces—the first piece goes to process 0, the second to process 1, the third to process 2, and so on. For example, suppose we're using a block distribution and process 0 has read in all of an n-component vector into `send_buf_p`. Then, process 0 will get the first `local_n` $= n/$`comm_sz` components, process 1 will get the next `local_n` components, and so on. Each process should pass its local vector as the `recv_buf_p` argument and the `recv_count` argument should be `local_n`. Both `send_type` and `recv_type` should be `MPI_DOUBLE` and `src_proc` should be 0. Perhaps surprisingly, `send_count` should also be `local_n`—`send_count` is the *amount of data going to each process*; it's *not* the amount of data in the memory referred to by `send_buf_p`. If we use a block distribution and `MPI_Scatter`, we can read in a vector using the function `Read_vector` shown in Program 3.9.

One point to note here is that `MPI_Scatter` sends the first block of `send_count` objects to process 0, the next block of `send_count` objects to process 1, and so on, so this approach to reading and distributing the input vectors will only be suitable if we're using a block distribution and n, the number of components in the vectors, is evenly divisible by `comm_sz`. We'll discuss a partial solution to dealing with a cyclic or block-cyclic distribution in Exercise 18. For a complete solution, see [23]. We'll look at dealing with the case in which n is not evenly divisible by `comm_sz` in Exercise 3.13.

```
1  void Read_vector(
2        double    local_a[]   /* out */,
3        int       local_n     /* in  */,
4        int       n           /* in  */,
5        char      vec_name[]  /* in  */,
6        int       my_rank     /* in  */,
7        MPI_Comm  comm        /* in  */) {
8
9     double* a = NULL;
10    int i;
11
12    if (my_rank == 0) {
13       a = malloc(n*sizeof(double));
14       printf("Enter the vector %s\n", vec_name);
15       for (i = 0; i < n; i++)
16          scanf("%lf", &a[i]);
17       MPI_Scatter(a, local_n, MPI_DOUBLE, local_a, local_n,
18             MPI_DOUBLE, 0, comm);
19       free(a);
20    } else {
21       MPI_Scatter(a, local_n, MPI_DOUBLE, local_a, local_n,
22             MPI_DOUBLE, 0, comm);
23    }
24 }  /* Read_vector */
```

Program 3.9: A function for reading and distributing a vector

3.4.8 Gather

Of course, our test program will be useless unless we can see the result of our vector addition, so we need to write a function for printing out a distributed vector. Our function can collect all of the components of the vector onto process 0, and then process 0 can print all of the components. The communication in this function can be carried out by `MPI_Gather`,

```
int MPI_Gather(
      void*         send_buf_p  /* in  */,
      int           send_count  /* in  */,
      MPI_Datatype  send_type   /* in  */,
      void*         recv_buf_p  /* out */,
      int           recv_count  /* in  */,
      MPI_Datatype  recv_type   /* in  */,
      int           dest_proc   /* in  */,
      MPI_Comm      comm        /* in  */);
```

The data stored in the memory referred to by `send_buf_p` on process 0 is stored in the first block in `recv_buf_p`, the data stored in the memory referred to by `send_buf_p` on process 1 is stored in the second block referred to by `recv_buf_p`, and so on. So, if we're using a block distribution, we can implement our distributed vector print function as shown in Program 3.10. Note that `recv_count` is the number of data items received from *each* process, not the total number of data items received.

```
1  void Print_vector(
2        double    local_b[]  /* in */,
3        int       local_n    /* in */,
4        int       n          /* in */,
5        char      title[]    /* in */,
6        int       my_rank    /* in */,
7        MPI_Comm  comm       /* in */) {
8
9     double* b = NULL;
10    int i;
11
12    if (my_rank == 0) {
13       b = malloc(n*sizeof(double));
14       MPI_Gather(local_b, local_n, MPI_DOUBLE, b, local_n,
15             MPI_DOUBLE, 0, comm);
16       printf("%s\n", title);
17       for (i = 0; i < n; i++)
18          printf("%f ", b[i]);
19       printf("\n");
20       free(b);
21    } else {
22       MPI_Gather(local_b, local_n, MPI_DOUBLE, b, local_n,
23             MPI_DOUBLE, 0, comm);
24    }
25 }  /* Print_vector */
```

Program 3.10: A function for printing a distributed vector

The restrictions on the use of `MPI_Gather` are similar to those on the use of `MPI_Scatter`: our print function will only work correctly with vectors using a block distribution in which each block has the same size.

3.4.9 Allgather

As a final example, let's look at how we might write an MPI function that multiplies a matrix by a vector. Recall that if $A = (a_{ij})$ is an $m \times n$ matrix and $\mathbf{x}$ is a vector with n components, then $\mathbf{y} = A\mathbf{x}$ is a vector with m components and we can find the ith component of y by forming the dot product of the ith row of A with $\mathbf{x}$:

$$y_i = a_{i0}x_0 + a_{i1}x_1 + a_{i2}x_2 + \cdots a_{i,n-1}x_{n-1}.$$

See Figure 3.11.

Thus, we might write pseudo-code for serial matrix multiplication as follows:

```
/* For each row of A */
for (i = 0; i < m; i++) {
   /* Form dot product of ith row with x */
   y[i] = 0.0;
   for (j = 0; j < n; j++)
      y[i] += A[i][j]*x[j];
}
```

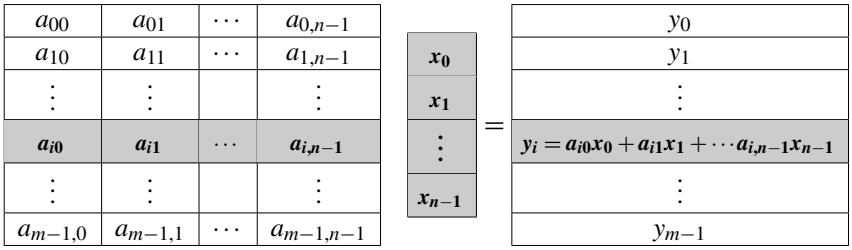


FIGURE 3.11

Matrix-vector multiplication

In fact, this could be actual C code. However, there are some peculiarities in the way that C programs deal with two-dimensional arrays (see Exercise 3.14), so C programmers frequently use one-dimensional arrays to "simulate" two-dimensional arrays. The most common way to do this is to list the rows one after another. For example, the two-dimensional array

$$\begin{pmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \end{pmatrix}$$

would be stored as the one-dimensional array

$$0\ 1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9\ 10\ 11.$$

In this example, if we start counting rows and columns from 0, then the element stored in row 2 and column 1 in the two-dimensional array (the 9), is located in position $2 \times 4 + 1 = 9$ in the one-dimensional array. More generally, if our array has n columns, when we use this scheme, we see that the element stored in row i and column j is located in position $i \times n + j$ in the one-dimensional array. Using this one-dimensional scheme, we get the C function shown in Program 3.11.

Now let's see how we might parallelize this function. An individual task can be the multiplication of an element of A by a component of $\mathbf{x}$ and the addition of this product into a component of $\mathbf{y}$. That is, each execution of the statement

```
y[i] += A[i*n+j]*x[j];
```

is a task. So we see that if `y[i]` is assigned to process q, then it would be convenient to also assign row i of `A` to process q. This suggests that we partition `A` by *rows*. We could partition the rows using a block distribution, a cyclic distribution, or a block-cyclic distribution. In MPI it's easiest to use a block distribution, so let's use a block distribution of the rows of `A`, and, as usual, assume that `comm_sz` evenly divides m, the number of rows.

We are distributing `A` by rows so that the computation of `y[i]` will have all of the needed elements of `A`, so we should distribute `y` by blocks. That is, if the ith row of

```
1  void Mat_vect_mult(
2        double  A[]  /* in  */,
3        double  x[]  /* in  */,
4        double  y[]  /* out */,
5        int     m    /* in  */,
6        int     n    /* in  */) {
7     int i, j;
8
9     for (i = 0; i < m; i++) {
10       y[i] = 0.0;
11       for (j = 0; j < n; j++)
12          y[i] += A[i*n+j]*x[j];
13    }
14 }  /* Mat_vect_mult */
```

Program 3.11: Serial matrix-vector multiplication

A, is assigned to process *q*, then the *i*th component of `y` should also be assigned to process *q*.

Now the computation of `y[i]` involves all the elements in the *i*th row of `A` and *all* the components of `x`, so we could minimize the amount of communication by simply assigning all of `x` to each process. However, in actual applications—especially when the matrix is square—it's often the case that a program using matrix-vector multiplication will execute the multiplication many times and the result vector **y** from one multiplication will be the input vector **x** for the next iteration. In practice, then, we usually assume that the distribution for `x` is the same as the distribution for `y`.

So if `x` has a block distribution, how can we arrange that each process has access to *all* the components of `x` before we execute the following loop?

```
for (j = 0; j < n; j++)
   y[i] += A[i*n+j]*x[j];
```

Using the collective communications we're already familiar with, we could execute a call to `MPI_Gather` followed by a call to `MPI_Bcast`. This would, in all likelihood, involve two tree-structured communications, and we may be able to do better by using a butterfly. So, once again, MPI provides a single function:

```
int MPI_Allgather(
      void*         send_buf_p  /* in */,
      int           send_count  /* in  */,
      MPI_Datatype  send_type   /* in  */,
      void*         recv_buf_p  /* out */,
      int           recv_count  /* in  */,
      MPI_Datatype  recv_type   /* in  */,
      MPI_Comm      comm        /* in  */);
```

This function concatenates the contents of each process' `send_buf_p` and stores this in each process' `recv_buf_p`. As usual, `recv_count` is the amount of data being

```
1  void Mat_vect_mult(
2        double    local_A[]  /* in  */,
3        double    local_x[]  /* in  */,
4        double    local_y[]  /* out */,
5        int       local_m    /* in  */,
6        int       n          /* in  */,
7        int       local_n    /* in  */,
8        MPI_Comm  comm       /* in  */) {
9     double* x;
10    int local_i, j;
11    int local_ok = 1;
12
13    x = malloc(n*sizeof(double));
14    MPI_Allgather(local_x, local_n, MPI_DOUBLE,
15          x, local_n, MPI_DOUBLE, comm);
16
17    for (local_i = 0; local_i < local_m; local_i++) {
18       local_y[local_i] = 0.0;
19       for (j = 0; j < n; j++)
20          local_y[local_i] += local_A[local_i*n+j]*x[j];
21    }
22    free(x);
23 }  /* Mat_vect_mult */
```

Program 3.12: An MPI matrix-vector multiplication function

received from *each* process, so in most cases, `recv_count` will be the same as `send_count`.

We can now implement our parallel matrix-vector multiplication function as shown in Program 3.12. If this function is called many times, we can improve performance by allocating `x` once in the calling function and passing it as an additional argument.

3.5 MPI DERIVED DATATYPES

In virtually all distributed-memory systems, communication can be *much* more expensive than local computation. For example, sending a **double** from one node to another will take far longer than adding two **double**s stored in the local memory of a node. Furthermore, the cost of sending a fixed amount of data in multiple messages is usually much greater than the cost of sending a single message with the same amount of data. For example, we would expect the following pair of `for` loops to be much slower than the single send/receive pair:

```
double x[1000];
. . .
if (my_rank == 0)
   for (i = 0; i < 1000; i++)
```

```
      MPI_Send(&x[i], 1, MPI_DOUBLE, 1, 0, comm);
else /* my_rank == 1 */
   for (i = 0; i < 1000; i++)
      MPI_Recv(&x[i], 1, MPI_DOUBLE, 0, 0, comm, &status);

if (my_rank == 0)
   MPI_Send(x, 1000, MPI_DOUBLE, 1, 0, comm);
else  /* my_rank == 1 */
   MPI_Recv(x, 1000, MPI_DOUBLE, 0, 0, comm, &status);
```

In fact, on one of our systems, the code with the loops of sends and receives takes nearly 50 times longer. On another system, the code with the loops takes more than 100 times longer. Thus, if we can reduce the total number of messages we send, we're likely to improve the performance of our programs.

MPI provides three basic approaches to consolidating data that might otherwise require multiple messages: the `count` argument to the various communication functions, derived datatypes, and `MPI_Pack/Unpack`. We've already seen the `count` argument—it can be used to group contiguous array elements into a single message. In this section we'll discuss one method for building derived datatypes. In the exercises, we'll take a look at some other methods for building derived datatypes and `MPI_Pack/Unpack`

In MPI, a **derived datatype** can be used to represent any collection of data items in memory by storing both the types of the items and their relative locations in memory. The idea here is that if a function that sends data knows the types and the relative locations in memory of a collection of data items, it can collect the items from memory before they are sent. Similarly, a function that receives data can distribute the items into their correct destinations in memory when they're received. As an example, in our trapezoidal rule program we needed to call `MPI_Bcast` three times: once for the left endpoint a, once for the right endpoint b, and once for the number of trapezoids n. As an alternative, we could build a single derived datatype that consists of two **double**s and one **int**. If we do this, we'll only need one call to `MPI_Bcast`. On process 0, a, b, and n will be sent with the one call, while on the other processes, the values will be received with the call.

Formally, a derived datatype consists of a sequence of basic MPI datatypes together with a *displacement* for each of the datatypes. In our trapezoidal rule example, suppose that on process 0 the variables `a`, `b`, and `n` are stored in memory locations with the following addresses:

Variable	Address
a	24
b	40
n	48

Then the following derived datatype could represent these data items:

$$\{(\texttt{MPI\_DOUBLE}, 0), (\texttt{MPI\_DOUBLE}, 16), (\texttt{MPI\_INT}, 24)\}.$$

The first element of each pair corresponds to the type of the data, and the second element of each pair is the displacement of the data element from the *beginning* of the type. We've assumed that the type begins with `a`, so it has displacement 0, and the other elements have displacements measured, in bytes, from `a`: `b` is $40 - 24 = 16$ bytes beyond the start of `a`, and `n` is $48 - 24 = 24$ bytes beyond the start of `a`.

We can use `MPI_Type_create_struct` to build a derived datatype that consists of individual elements that have different basic types:

```
int MPI_Type_create_struct(
      int            count                       /* in  */,
      int            array_of_blocklengths[]     /* in  */,
      MPI_Aint       array_of_displacements[]    /* in  */,
      MPI_Datatype   array_of_types[]            /* in  */,
      MPI_Datatype*  new_type_p                  /* out */);
```

The argument `count` is the number of elements in the datatype, so for our example, it should be three. Each of the array arguments should have `count` elements. The first array, `array_of_block_lengths`, allows for the possibility that the individual data items might be arrays or subarrays. If, for example, the first element were an array containing five elements, we would have

```
array_of_blocklengths[0] = 5;
```

However, in our case, none of the elements is an array, so we can simply define

```
int array_of_blocklengths[3] = {1, 1, 1};
```

The third argument to `MPI_Type_create_struct`, `array_of_displacements`, specifies the displacements, in bytes, from the start of the message. So we want

```
array_of_displacements[] = {0, 16, 24};
```

To find these values, we can use the function `MPI_Get_address`:

```
int MPI_Get_address(
      void*      location_p  /* in  */,
      MPI_Aint*  address_p   /* out */);
```

It returns the address of the memory location referenced by `location_p`. The special type `MPI_Aint` is an integer type that is big enough to store an address on the system. Thus, in order to get the values in `array_of_displacements`, we can use the following code:

```
MPI_Aint a_addr, b_addr, n_addr;

MPI_Get_address(&a, &a_addr);
array_of_displacements[0] = 0;
MPI_Get_address(&b, &b_addr);
array_of_displacements[1] = b_addr - a_addr;
MPI_Get_address(&n, &n_addr);
array_of_displacements[2] = n_addr - a_addr;
```

The array_of_datatypes should store the MPI datatypes of the elements, so we can just define

```
MPI_Datatype array_of_types[3] = {MPI_DOUBLE, MPI_DOUBLE, MPI_INT};
```

With these initializations, we can build the new datatype with the call

```
MPI_Datatype input_mpi_t;
. . .
MPI_Type_create_struct(3, array_of_blocklengths,
      array_of_displacements, array_of_types,
      &input_mpi_t);
```

Before we can use input_mpi_t in a communication function, we must first **commit** it with a call to

```
int MPI_Type_commit(MPI_Datatype*  new_mpi_t_p  /* in/out */);
```

This allows the MPI implementation to optimize its internal representation of the datatype for use in communication functions.

Now, in order to use new_mpi_t, we make the following call to MPI_Bcast on each process:

```
MPI_Bcast(&a, 1, input_mpi_t, 0, comm);
```

So we can use input_mpi_t just as we would use one of the basic MPI datatypes.

In constructing the new datatype, it's likely that the MPI implementation had to allocate additional storage internally. Therefore, when we're through using the new type, we can free any additional storage used with a call to

```
int MPI_Type_free(MPI_Datatype*  old_mpi_t_p  /* in/out */);
```

We used the steps outlined here to define a Build_mpi_type function that our Get_input function can call. The new function and the updated Get_input function are shown in Program 3.13.

3.6 PERFORMANCE EVALUATION OF MPI PROGRAMS

Let's take a look at the performance of the matrix-vector multiplication program. For the most part we write parallel programs because we expect that they'll be faster than a serial program that solves the same problem. How can we verify this? We spent some time discussing this in Section 2.6, so we'll start by recalling some of the material we learned there.

3.6.1 Taking timings

We're usually not interested in the time taken from the start of program execution to the end of program execution. For example, in the matrix-vector multiplication, we're not interested in the time it takes to type in the matrix or print out the product.

```
void Build_mpi_type(
      double*         a_p             /* in  */,
      double*         b_p             /* in  */,
      int*            n_p             /* in  */,
      MPI_Datatype*   input_mpi_t_p   /* out */) {

   int array_of_blocklengths[3] = {1, 1, 1};
   MPI_Datatype array_of_types[3] = {MPI_DOUBLE, MPI_DOUBLE, MPI_INT};
   MPI_Aint a_addr, b_addr, n_addr;
   MPI_Aint array_of_displacements[3] = {0};

   MPI_Get_address(a_p, &a_addr);
   MPI_Get_address(b_p, &b_addr);
   MPI_Get_address(n_p, &n_addr);
   array_of_displacements[1] = b_addr-a_addr;
   array_of_displacements[2] = n_addr-a_addr;
   MPI_Type_create_struct(3, array_of_blocklengths,
         array_of_displacements, array_of_types,
         input_mpi_t_p);
   MPI_Type_commit(input_mpi_t_p);
}  /* Build_mpi_type */

void Get_input(int my_rank, int comm_sz, double* a_p, double* b_p,
      int* n_p) {
   MPI_Datatype input_mpi_t;

   Build_mpi_type(a_p, b_p, n_p, &input_mpi_t);

   if (my_rank == 0) {
      printf("Enter a, b, and n\n");
      scanf("%lf %lf %d", a_p, b_p, n_p);
   }
   MPI_Bcast(a_p, 1, input_mpi_t, 0, MPI_COMM_WORLD);

   MPI_Type_free(&input_mpi_t);
}  /* Get_input */
```

Program 3.13: The `Get_input` function with a derived datatype

We're only interested in the time it takes to do the actual multiplication, so we need to modify our source code by adding in calls to a function that will tell us the amount of time that elapses from the beginning to the end of the actual matrix-vector multiplication. MPI provides a function, `MPI_Wtime`, that returns the number of seconds that have elapsed since some time in the past:

```
double MPI_Wtime(void);
```

Thus, we can time a block of MPI code as follows:

```
double start, finish;
. . .
```

```
start = MPI_Wtime();
/* Code to be timed */
. . .
finish = MPI_Wtime();
printf("Proc %d > Elapsed time = %e seconds\n"
      my_rank, finish-start);
```

In order to time serial code, it's not necessary to link in the MPI libraries. There is a POSIX library function called `gettimeofday` that returns the number of microseconds that have elapsed since some point in the past. The syntax details aren't too important. There's a C macro `GET_TIME` defined in the header file `timer.h` that can be downloaded from the book's website. This macro should be called with a **double** argument:

```
#include "timer.h"
. . .
double now;
. . .
GET_TIME(now);
```

After executing this macro, `now` will store the number of seconds since some time in the past. We can get the elapsed time of serial code with microsecond resolution by executing

```
#include "timer.h"
. . .
double start, finish;
. . .
GET_TIME(start);
/* Code to be timed */
. . .
GET_TIME(finish);
printf("Elapsed time = %e seconds\n", finish-start);
```

One point to stress here: `GET_TIME` is a macro, so the code that defines it is inserted directly into your source code by the preprocessor. Hence, it can operate directly on its argument, and the argument is a **double**, *not* a pointer to a **double**. A final note in this connection: Since `timer.h` is not in the system include file directory, it's necessary to tell the compiler where to find it if it's not in the directory where you're compiling. For example, if it's in the directory `/home/peter/my_include`, the following command can be used to compile a serial program that uses `GET_TIME`:

```
$ gcc -g -Wall -I/home/peter/my_include -o <executable>
      <source_code.c>
```

Both `MPI_Wtime` and `GET_TIME` return *wall clock* time. Recall that a timer like the C `clock` function returns CPU time—the time spent in user code, library functions, and operating system code. It doesn't include idle time, which can be a significant part of parallel run time. For example, a call to `MPI_Recv` may spend a significant amount of time waiting for the arrival of a message. Wall clock time, on the other hand, gives total elapsed time, so it includes idle time.

There are still a few remaining issues. First, as we've described it, our parallel program will report `comm_sz` times, one for each process. We would like to have it report a single time. Ideally, all of the processes would start execution of the matrix-vector multiplication at the same time, and then, we would report the time that elapsed when the last process finished. In other words, the parallel execution time would be the time it took the "slowest" process to finish. We can't get exactly this time because we can't insure that all the processes start at the same instant. However, we can come reasonably close. The MPI collective communication function `MPI_Barrier` insures that no process will return from calling it until every process in the communicator has started calling it. It's syntax is

```
int MPI_Barrier(MPI_Comm   comm   /* in */);
```

The following code can be used to time a block of MPI code and report a single elapsed time:

```
double local_start, local_finish, local_elapsed, elapsed;
. . .
MPI_Barrier(comm);
local_start = MPI_Wtime();
/* Code to be timed */
. . .
local_finish = MPI_Wtime();
local_elapsed = local_finish - local_start;
MPI_Reduce(&local_elapsed, &elapsed, 1, MPI_DOUBLE,
   MPI_MAX, 0, comm);

if (my_rank == 0)
   printf("Elapsed time = %e seconds\n", elapsed);
```

Note that the call to `MPI_Reduce` is using the `MPI_MAX` operator; it finds the largest of the input arguments `local_elapsed`.

As we noted in Chapter 2, we also need to be aware of variability in timings: when we run a program several times, we're likely to see a substantial variation in the times. This will be true even if for each run we use the same input, the same number of processes, and the same system. This is because the interaction of the program with the rest of the system, especially the operating system, is unpredictable. Since this interaction will almost certainly not make the program run faster than it would run on a "quiet" system, we usually report the *minimum* run-time rather than the mean or median. (For further discussion of this, see [5].)

Finally, when we run an MPI program on a hybrid system in which the nodes are multicore processors, we'll only run one MPI process on each node. This may reduce contention for the interconnect and result in somewhat better run-times. It may also reduce variability in run-times.

3.6.2 Results

The results of timing the matrix-vector multiplication program are shown in Table 3.5. The input matrices were square. The times shown are in milliseconds,

Table 3.5 Run-Times of Serial and Parallel Matrix-Vector Multiplication (times are in milliseconds)

	Order of Matrix				
comm_sz	*1024*	*2048*	*4096*	*8192*	*16,384*
1	4.1	16.0	64.0	270	1100
2	2.3	8.5	33.0	140	560
4	2.0	5.1	18.0	70	280
8	1.7	3.3	9.8	36	140
16	1.7	2.6	5.9	19	71

and we've rounded each time to two significant digits. The times for comm_sz = 1 are the run-times of the serial program running on a single core of the distributed-memory system. Not surprisingly, if we fix comm_sz, and increase n, the order of the matrix, the run-times increase. For relatively small numbers of processes, doubling n results in roughly a four-fold increase in the run-time. However, for large numbers of processes, this formula breaks down.

If we fix n and increase comm_sz, the run-times usually decrease. In fact, for large values of n, doubling the number of processes roughly halves the overall run-time. However, for small n, there is very little benefit in increasing comm_sz. In fact, in going from 8 to 16 processes when $n = 1024$, the overall run time is unchanged.

These timings are fairly typical of parallel run-times—as we increase the problem size, the run-times increase, and this is true regardless of the number of processes. The rate of increase can be fairly constant (e.g., the one-process times) or it can vary wildly (e.g., the 16-process times). As we increase the number of processes, the run-times typically decrease for a while. However, at some point, the run-times can actually start to get worse. The closest we came to this behavior was going from 8 to 16 processes when the matrix had order 1024.

The explanation for this is that there is a fairly common relation between the run-times of serial programs and the run-times of corresponding parallel programs. Recall that we denote the serial run-time by T_{serial}. Since it typically depends on the size of the input, n, we'll frequently denote it as $T_{\text{serial}}(n)$. Also recall that we denote the parallel run-time by T_{parallel}. Since it depends on both the input size, n, and the number of processes, comm_sz = p, we'll frequently denote it as $T_{\text{parallel}}(n,p)$. As we noted in Chapter 2, it's often the case that the parallel program will divide the work of the serial program among the processes, and add in some overhead time, which we denoted T_{overhead}:

$$T_{\text{parallel}}(n,p) = T_{\text{serial}}(n)/p + T_{\text{overhead}}.$$

In MPI programs, the parallel overhead typically comes from communication, and it can depend on both the problem size and the number of processes.

It's not too hard to see that this formula applies to our matrix-vector multiplication program. The heart of the serial program is the pair of nested **for** loops:

```
for (i = 0; i < m; i++) {
   y[i] = 0.0;
   for (j = 0; j < n; j++)
      y[i] += A[i*n+j]*x[j];
}
```

If we only count floating point operations, the inner loop carries out n multiplications and n additions, for a total of $2n$ floating point operations. Since we execute the inner loop m times, the pair of loops executes a total of $2mn$ floating point operations. So when $m = n$,

$$T_{\text{serial}}(n) \approx an^2$$

for some constant a. (The symbol $\approx$ means "is approximately equal to.")

If the serial program multiplies an $n \times n$ matrix by an n-dimensional vector, then each process in the parallel program multiplies an $n/p \times n$ matrix by an n-dimensional vector. The *local* matrix-vector multiplication part of the parallel program therefore executes n^2/p floating point operations. Thus, it appears that this *local* matrix-vector multiplication reduces the work per process by a factor of p.

However, the parallel program also needs to complete a call to `MPI_Allgather` before it can carry out the local matrix-vector multiplication. In our example, it appears that

$$T_{\text{parallel}}(n,p) = T_{\text{serial}}(n)/p + T_{\text{allgather}}.$$

Furthermore, in light of our timing data, it appears that for smaller values of p and larger values of n, the dominant term in our formula is $T_{\text{serial}}(n)/p$. To see this, observe first that for small p (e.g., $p = 2,4$), doubling p roughly halves the overall run-time. For example,

$$T_{\text{serial}}(4096) = 1.9 \times T_{\text{parallel}}(4096,2)$$

$$T_{\text{serial}}(8192) = 1.9 \times T_{\text{parallel}}(8192,2)$$

$$T_{\text{parallel}}(8192,2) = 2.0 \times T_{\text{parallel}}(8192,4)$$

$$T_{\text{serial}}(16,384) = 2.0 \times T_{\text{parallel}}(16,384,2)$$

$$T_{\text{parallel}}(16,384,2) = 2.0 \times T_{\text{parallel}}(16,384,4)$$

Also, if we fix p at a small value (e.g., $p = 2,4$), then increasing n seems to have approximately the same effect as increasing n for the serial program. For example,

$$T_{\text{serial}}(4096) = 4.0 \times T_{\text{serial}}(2048)$$

$$T_{\text{parallel}}(4096,2) = 3.9 \times T_{\text{parallel}}(2048,2)$$

$$T_{\text{parallel}}(4096,4) = 3.5 \times T_{\text{parallel}}(2048,4)$$

$$T_{\text{serial}}(8192) = 4.2 \times T_{\text{serial}}(4096)$$

$$T_{\text{parallel}}(8192,2) = 4.2 \times T_{\text{parallel}}(4096,2)$$

$$T_{\text{parallel}}(8192,4) = 3.9 \times T_{\text{parallel}}(8192,4)$$

These observations suggest that the parallel run-times are behaving much as the run-times of the serial program—that is, $T_{\text{parallel}}(n,p)$ is approximately $T_{\text{serial}}(n)/p$—so the overhead $T_{\text{allgather}}$ has little effect on the performance.

On the other hand, for small n and large p these patterns break down. For example,

$$T_{\text{parallel}}(1024,8) = 1.0 \times T_{\text{parallel}}(1024,16)$$

$$T_{\text{parallel}}(2048,16) = 1.5 \times T_{\text{parallel}}(1024,16)$$

Thus, it appears that for small n and large p, the dominant term in our formula for T_{parallel} is $T_{\text{allgather}}$.

3.6.3 Speedup and efficiency

Recall that the most widely used measure of the relation between the serial and the parallel run-times is the **speedup**. It's just the ratio of the serial run-time to the parallel run-time:

$$S(n,p) = \frac{T_{\text{serial}}(n)}{T_{\text{parallel}}(n,p)}.$$

The ideal value for $S(n,p)$ is p. If $S(n,p) = p$, then our parallel program with `comm_sz` $= p$ processes is running p times faster than the serial program. In practice, this speedup, sometimes called **linear speedup**, is rarely achieved. Our matrix-vector multiplication program got the speedups shown in Table 3.6. For small p and large n, our program obtained nearly linear speedup. On the other hand, for large p and small n, the speedup was considerably less than p. The worst case was $n = 1024$ and $p = 16$, when we only managed a speedup of 2.4.

Table 3.6 Speedups of Parallel Matrix-Vector Multiplication

	Order of Matrix				
comm_sz	*1024*	*2048*	*4096*	*8192*	*16,384*
1	1.0	1.0	1.0	1.0	1.0
2	1.8	1.9	1.9	1.9	2.0
4	2.1	3.1	3.6	3.9	3.9
8	2.4	4.8	6.5	7.5	7.9
16	2.4	6.2	10.8	14.2	15.5

Table 3.7 Efficiencies of Parallel Matrix-Vector Multiplication

	Order of Matrix				
comm_sz	*1024*	*2048*	*4096*	*8192*	*16,384*
1	1.00	1.00	1.00	1.00	1.00
2	0.89	0.94	0.97	0.96	0.98
4	0.51	0.78	0.89	0.96	0.98
8	0.30	0.61	0.82	0.94	0.98
16	0.15	0.39	0.68	0.89	0.97

Also recall that another widely used measure of parallel performance is parallel **efficiency**. This is "per process" speedup:

$$E(n,p) = \frac{S(n,p)}{p} = \frac{T_{\text{serial}}(n)}{p \times T_{\text{parallel}}(n,p)}.$$

Linear speedup corresponds to a parallel efficiency of $p/p = 1.0$, and, in general, we expect that our efficiencies will be less than 1.

The efficiencies for the matrix-vector multiplication program are shown in Table 3.7. Once again, for small p and large n our parallel efficiencies are near linear, and for large p and small n, they are very far from linear.

3.6.4 Scalability

Our parallel matrix-vector multiplication program doesn't come close to obtaining linear speedup for small n and large p. Does this mean that it's not a good program? Many computer scientists answer this question by looking at the "scalability" of the program. Recall that very roughly speaking, a program is **scalable** if the problem size can be increased at a rate so that the efficiency doesn't decrease as the number of processes increase.

The problem with this definition is the phrase "the problem size can be increased at a rate . . . " Consider two parallel programs: program A and program B. Suppose that if $p \geq 2$, the efficiency of program A is 0.75, regardless of problem size. Also suppose that the efficiency of program B is $n/(625p)$, provided $p \geq 2$ and $1000 \leq n \leq 625p$. Then according to our "definition," both programs are scalable. For program A, the rate of increase needed to maintain constant efficiency is 0, while for program B if we increase n at the same rate as we increase p, we'll maintain a constant efficiency. For example, if $n = 1000$ and $p = 2$, the efficiency of B is 0.80. If we then double p to 4 and we leave the problem size at $n = 1000$, the efficiency will drop to 0.40, but if we also double the problem size to $n = 2000$, the efficiency will remain constant at 0.80. Program A is thus *more* scalable than B, but both satisfy our definition of scalability.

Looking at our table of parallel efficiencies (Table 3.7), we see that our matrix-vector multiplication program definitely doesn't have the same scalability as program *A*: in almost every case when p is increased, the efficiency decreases. On the other hand, the program is somewhat like program *B*: if $p \geq 2$ and we increase both p and n by a factor of 2, the parallel efficiency, for the most part, actually increases. Furthermore, the only exceptions occur when we increase p from 2 to 4, and when computer scientists discuss scalability, they're usually interested in large values of p. When p is increased from 4 to 8 or from 8 to 16, our efficiency always increases when we increase n by a factor of 2.

Recall that programs that can maintain a constant efficiency without increasing the problem size are sometimes said to be **strongly scalable**. Programs that can maintain a constant efficiency if the problem size increases at the same rate as the number of processes are sometimes said to be **weakly scalable**. Program *A* is strongly scalable, and program *B* is weakly scalable. Furthermore, our matrix-vector multiplication program is also apparently weakly scalable.

3.7 A PARALLEL SORTING ALGORITHM

What do we mean by a parallel sorting algorithm in a distributed-memory environment? What would its "input" be and what would its "output" be? The answers depend on where the keys are stored. We can start or finish with the keys distributed among the processes or assigned to a single process. In this section we'll look at an algorithm that starts and finishes with the keys distributed among the processes. In Programming Assignment 3.8 we'll look at an algorithm that finishes with the keys assigned to a single process.

If we have a total of n keys and $p = $ `comm_sz` processes, our algorithm will start and finish with n/p keys assigned to each process. (As usual, we'll assume n is evenly divisible by p.) At the start, there are no restrictions on which keys are assigned to which processes. However, when the algorithm terminates,

- the keys assigned to each process should be sorted in (say) increasing order, and
- if $0 \leq q < r < p$, then each key assigned to process q should be less than or equal to every key assigned to process r.

So if we lined up the keys according to process rank—keys from process 0 first, then keys from process 1, and so on—then the keys would be sorted in increasing order. For the sake of explicitness, we'll assume our keys are ordinary **int**s.

3.7.1 Some simple serial sorting algorithms

Before starting, let's look at a couple of simple serial sorting algorithms. Perhaps the best known serial sorting algorithm is bubble sort (see Program 3.14). The array `a` stores the unsorted keys when the function is called, and the sorted keys when the function returns. The number of keys in `a` is `n`. The algorithm proceeds by comparing

```
1  void Bubble_sort(
2         int  a[]  /* in/out */,
3         int  n    /* in     */) {
4     int list_length, i, temp;
5
6     for (list_length = n; list_length >= 2; list_length--)
7        for (i = 0; i < list_length-1; i++)
8           if (a[i] > a[i+1]) {
9              temp = a[i];
10             a[i] = a[i+1];
11             a[i+1] = temp;
12          }
13
14 }  /* Bubble_sort */
```

Program 3.14: Serial bubble sort

the elements of the list `a` pairwise: `a[0]` is compared to `a[1]`, `a[1]` is compared to `a[2]`, and so on. Whenever a pair is out of order, the entries are swapped, so in the first pass through the outer loop, when `list_length = n`, the largest value in the list will be moved into `a[n-1]`. The next pass will ignore this last element and it will move the next-to-the-largest element into `a[n-2]`. Thus, as `list_length` decreases, successively more elements get assigned to their final positions in the sorted list.

There isn't much point in trying to parallelize this algorithm because of the inherently sequential ordering of the comparisons. To see this, suppose that `a[i-1] = 9`, `a[i] = 5`, and `a[i+1] = 7`. The algorithm will first compare 9 and 5 and swap them, it will then compare 9 and 7 and swap them, and we'll have the sequence $5, 7, 9$. If we try to do the comparisons out of order, that is, if we compare the 5 and 7 first and then compare the 9 and 5, we'll wind up with the sequence $5, 9, 7$. Therefore, the order in which the "compare-swaps" take place is essential to the correctness of the algorithm.

A variant of bubble sort known as **odd-even transposition sort** has considerably more opportunities for parallelism. The key idea is to "decouple" the compare-swaps. The algorithm consists of a sequence of *phases,* of two different types. During *even* phases, compare-swaps are executed on the pairs

$$(a[0], a[1]), (a[2], a[3]), (a[4], a[5]), \ldots,$$

and during *odd* phases, compare-swaps are executed on the pairs

$$(a[1], a[2]), (a[3], a[4]), (a[5], a[6]), \ldots.$$

Here's a small example:

Start: $5, 9, 4, 3$
Even phase: Compare-swap $(5, 9)$ and $(4, 3)$, getting the list $5, 9, 3, 4$.
Odd phase: Compare-swap $(9, 3)$, getting the list $5, 3, 9, 4$.
Even phase: Compare-swap $(5, 3)$ and $(9, 4)$, getting the list $3, 5, 4, 9$.
Odd phase: Compare-swap $(5, 4)$, getting the list $3, 4, 5, 9$.

This example required four phases to sort a four-element list. In general, it may require fewer phases, but the following theorem guarantees that we can sort a list of n elements in at most n phases:

Theorem. *Suppose A is a list with n keys, and A is the input to the odd-even transposition sort algorithm. Then, after n phases A will be sorted.*

Program 3.15 shows code for a serial odd-even transposition sort function.

```
1  void Odd_even_sort(
2        int  a[]  /* in/out */,
3        int  n    /* in     */) {
4     int phase, i, temp;
5
6     for (phase = 0; phase < n; phase++)
7        if (phase % 2 == 0) { /* Even phase */
8           for (i = 1; i < n; i += 2)
9              if (a[i-1] > a[i]) {
10                temp = a[i];
11                a[i] = a[i-1];
12                a[i-1] = temp;
13             }
14       } else { /* Odd phase */
15          for (i = 1; i < n-1; i += 2)
16             if (a[i] > a[i+1]) {
17                temp = a[i];
18                a[i] = a[i+1];
19                a[i+1] = temp;
20             }
21       }
22 }  /* Odd_even_sort */
```

Program 3.15: Serial odd-even transposition sort

3.7.2 Parallel odd-even transposition sort

It should be clear that odd-even transposition sort has considerably more opportunities for parallelism than bubble sort, because all of the compare-swaps in a single phase can happen simultaneously. Let's try to exploit this.

There are a number of possible ways to apply Foster's methodology. Here's one:

- *Tasks:* Determine the value of `a[i]` at the end of phase j.
- *Communications:* The task that's determining the value of `a[i]` needs to communicate with either the task determining the value of `a[i-1]` or `a[i+1]`. Also the value of `a[i]` at the end of phase j needs to be available for determining the value of `a[i]` at the end of phase $j+1$.

This is illustrated in Figure 3.12, where we've labeled the tasks determining the value of `a[i]` with `a[i]`.

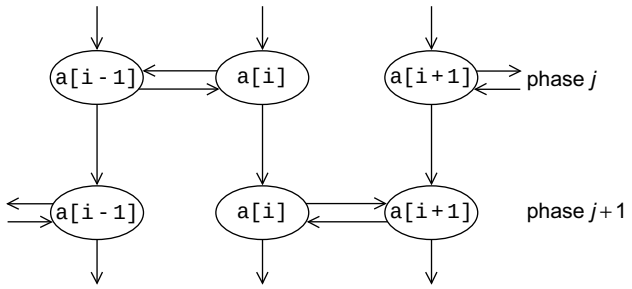


FIGURE 3.12

Communications among tasks in an odd-even sort. Tasks determining `a[i]` are labeled with `a[i]`.

Now recall that when our sorting algorithm starts and finishes execution, each process is assigned n/p keys. In this case our aggregation and mapping are at least partially specified by the description of the problem. Let's look at two cases.

When $n = p$, Figure 3.12 makes it fairly clear how the algorithm should proceed. Depending on the phase, process i can send its current value, `a[i]`, either to process $i - 1$ or process $i + 1$. At the same time, it should receive the value stored on process $i - 1$ or process $i + 1$, respectively, and then decide which of the two values it should store as `a[i]` for the next phase.

However, it's unlikely that we'll actually want to apply the algorithm when $n = p$, since we're unlikely to have more than a few hundred or a few thousand processors at our disposal, and sorting a few thousand values is usually a fairly trivial matter for a single processor. Furthermore, even if we do have access to thousands or even millions of processors, the added cost of sending and receiving a message for each compare-exchange will slow the program down so much that it will be useless. Remember that the cost of communication is usually much greater than the cost of "local" computation—for example, a compare-swap.

How should this be modified when each process is storing $n/p > 1$ elements? (Recall that we're assuming that n is evenly divisible by p.) Let's look at an example. Suppose we have $p = 4$ processes and $n = 16$ keys assigned, as shown in Table 3.8. In the first place, we can apply a fast serial sorting algorithm to the keys assigned to each process. For example, we can use the C library function `qsort` on each process to sort the local keys. Now if we had one element per process, 0 and 1 would exchange elements, and 2 and 3 would exchange. So let's try this: Let's have 0 and 1 exchange *all* their elements and 2 and 3 exchange all of theirs. Then it would seem natural for 0 to keep the four smaller elements and 1 to keep the larger. Similarly, 2 should keep the smaller and 3 the larger. This gives us the situation shown in the third row of the the table. Once again, looking at the one element per process case, in phase 1, processes 1 and 2 exchange their elements and processes 0 and 3 are idle. If process 1 keeps the smaller and 2 the larger elements, we get the distribution shown in the fourth row. Continuing this process for two more phases results in a sorted list. That is, each process' keys are stored in increasing order, and if $q < r$,

Table 3.8 Parallel Odd-Even Transposition Sort

Time	Process 0	Process 1	Process 2	Process 3
Start	15, 11, 9, 16	3, 14, 8, 7	4, 6, 12, 10	5, 2, 13, 1
After Local Sort	9, 11, 15, 16	3, 7, 8, 14	4, 6, 10, 12	1, 2, 5, 13
After Phase 0	3, 7, 8, 9	11, 14, 15, 16	1, 2, 4, 5	6, 10, 12, 13
After Phase 1	3, 7, 8, 9	1, 2, 4, 5	11, 14, 15, 16	6, 10, 12, 13
After Phase 2	1, 2, 3, 4	5, 7, 8, 9	6, 10, 11, 12	13, 14, 15, 16
After Phase 3	1, 2, 3, 4	5, 6, 7, 8	9, 10, 11, 12	13, 14, 15, 16

then the keys assigned to process q are less than or equal to the keys assigned to process r.

In fact, our example illustrates the worst-case performance of this algorithm:

Theorem. *If parallel odd-even transposition sort is run with p processes, then after p phases, the input list will be sorted.*

The parallel algorithm is clear to a human computer:

```
Sort local keys;
for (phase = 0; phase < comm_sz; phase++) {
   partner = Compute_partner(phase, my_rank);
   if (I'm not idle) {
      Send my keys to partner;
      Receive keys from partner;
      if (my_rank < partner)
         Keep smaller keys;
      else
         Keep larger keys;
   }
}
```

However, there are some details that we need to clear up before we can convert the algorithm into an MPI program.

First, how do we compute the partner rank? And what is the partner rank when a process is idle? If the phase is even, then odd-ranked partners exchange with `my_rank−1` and even-ranked partners exchange with `my_rank+1`. In odd phases, the calculations are reversed. However, these calculations can return some invalid ranks: if `my_rank = 0` or `my_rank = comm_sz−1`, the partner rank can be −1 or `comm_sz`. But when either `partner = −1` or `partner = comm_sz`, the process should be idle. We can use the rank computed by `Compute_partner` to determine whether a process is idle:

```
if (phase % 2 == 0)      /* Even phase */
   if (my_rank % 2 != 0)     /* Odd rank */
      partner = my_rank - 1;
   else                      /* Even rank */
      partner = my_rank + 1;
```

```
else                    /* Odd phase */
   if (my_rank % 2 != 0)     /* Odd rank */
      partner = my_rank + 1;
   else                      /* Even rank */
      partner = my_rank - 1;
if (partner == -1 || partner == comm_sz)
   partner = MPI_PROC_NULL;
```

`MPI_PROC_NULL` is a constant defined by MPI. When it's used as the source or destination rank in a point-to-point communication, no communication will take place and the call to the communication will simply return.

3.7.3 Safety in MPI programs

If a process is not idle, we might try to implement the communication with a call to `MPI_Send` and a call to `MPI_Recv`:

```
MPI_Send(my_keys, n/comm_sz, MPI_INT, partner, 0, comm);
MPI_Recv(temp_keys, n/comm_sz, MPI_INT, partner, 0, comm,
      MPI_STATUS_IGNORE);
```

This, however, might result in the programs' hanging or crashing. Recall that the MPI standard allows `MPI_Send` to behave in two different ways: it can simply copy the message into an MPI-managed buffer and return, or it can block until the matching call to `MPI_Recv` starts. Furthermore, many implementations of MPI set a threshold at which the system switches from buffering to blocking. That is, messages that are relatively small will be buffered by `MPI_Send`, but for larger messages, it will block. If the `MPI_Send` executed by each process blocks, no process will be able to start executing a call to `MPI_Recv`, and the program will hang or **deadlock**, that is, each process is blocked waiting for an event that will never happen.

A program that relies on MPI-provided buffering is said to be **unsafe**. Such a program may run without problems for various sets of input, but it may hang or crash with other sets. If we use `MPI_Send` and `MPI_Recv` in this way, our program will be unsafe, and it's likely that for small values of n the program will run without problems, while for larger values of n, it's likely that it will hang or crash.

There are a couple of questions that arise here:

1. In general, how can we tell if a program is safe?
2. How can we modify the communication in the parallel odd-even sort program so that it is safe?

To answer the first question, we can use an alternative to `MPI_Send` defined by the MPI standard. It's called `MPI_Ssend`. The extra "s" stands for *synchronous* and `MPI_Ssend` is guaranteed to block until the matching receive starts. So, we can check whether a program is safe by replacing the calls to `MPI_Send` with calls to `MPI_Ssend`. If the program doesn't hang or crash when it's run with appropriate input and `comm_sz`, then the original program was safe. The arguments to `MPI_Ssend` are the same as the arguments to `MPI_Send`:

```
int MPI_Ssend(
      void*         msg_buf_p     /* in */,
      int           msg_size      /* in */,
      MPI_Datatype  msg_type      /* in */,
      int           dest          /* in */,
      int           tag           /* in */,
      MPI_Comm      communicator  /* in */);
```

The answer to the second question is that the communication must be restructured. The most common cause of an unsafe program is multiple processes simultaneously first sending to each other and then receiving. Our exchanges with partners is one example. Another example is a "ring pass," in which each process q sends to the process with rank $q+1$, except that process comm_sz -1 sends to 0:

```
MPI_Send(msg, size, MPI_INT, (my_rank+1) % comm_sz, 0, comm);
MPI_Recv(new_msg, size, MPI_INT, (my_rank+comm_sz-1) % comm_sz,
      0, comm, MPI_STATUS_IGNORE).
```

In both settings, we need to restructure the communications so that some of the processes receive before sending. For example, the preceding communications could be restructured as follows:

```
if (my_rank % 2 == 0) {
   MPI_Send(msg, size, MPI_INT, (my_rank+1) % comm_sz, 0, comm);
   MPI_Recv(new_msg, size, MPI_INT, (my_rank+comm_sz-1) % comm_sz,
         0, comm, MPI_STATUS_IGNORE).
} else {
   MPI_Recv(new_msg, size, MPI_INT, (my_rank+comm_sz-1) % comm_sz,
         0, comm, MPI_STATUS_IGNORE).
   MPI_Send(msg, size, MPI_INT, (my_rank+1) % comm_sz, 0, comm);
}
```

It's fairly clear that this will work if comm_sz is even. If, say, comm_sz $= 4$, then processes 0 and 2 will first send to 1 and 3, respectively, while processes 1 and 3 will receive from 0 and 2, respectively. The roles are reversed for the next send-receive pairs: processes 1 and 3 will send to 2 and 0, respectively, while 2 and 0 will receive from 1 and 3.

However, it may not be clear that this scheme is also safe if comm_sz is odd (and greater than 1). Suppose, for example, that comm_sz $= 5$. Then, Figure 3.13 shows a possible sequence of events. The solid arrows show a completed communication, and the dashed arrows show a communication waiting to complete.

MPI provides an alternative to scheduling the communications ourselves—we can call the function MPI_Sendrecv:

```
int MPI_Sendrecv(
      void*         send_buf_p     /* in  */,
      int           send_buf_size  /* in  */,
      MPI_Datatype  send_buf_type  /* in  */,
      int           dest           /* in  */,
      int           send_tag       /* in  */,
```

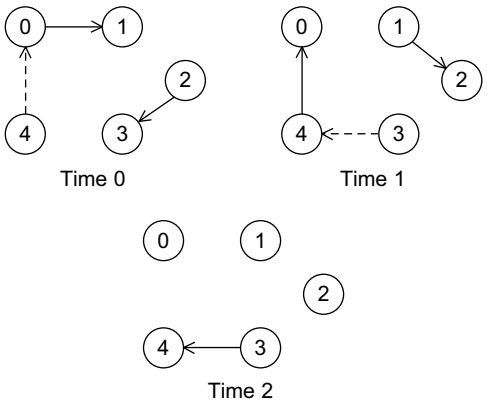


FIGURE 3.13

Safe communication with five processes

```
      void*          recv_buf_p     /* out */,
      int            recv_buf_size  /* in  */,
      MPI_Datatype   recv_buf_type  /* in  */,
      int            source         /* in  */,
      int            recv_tag       /* in  */,
      MPI_Comm       communicator   /* in  */,
      MPI_Status*    status_p       /* in  */);
```

This function carries out a blocking send and a receive in a single call. The `dest` and the `source` can be the same or different. What makes it especially useful is that the MPI implementation schedules the communications so that the program won't hang or crash. The complex code we used earlier—the code that checks whether the process rank is odd or even—can be replaced with a single call to `MPI_Sendrecv`. If it happens that the send and the receive buffers should be the same, MPI provides the alternative:

```
int MPI_Sendrecv_replace(
      void*          buf_p          /* in/out */,
      int            buf_size       /* in     */,
      MPI_Datatype   buf_type       /* in     */,
      int            dest           /* in     */,
      int            send_tag       /* in     */,
      int            source         /* in     */,
      int            recv_tag       /* in     */,
      MPI_Comm       communicator   /* in     */,
      MPI_Status*    status_p       /* in     */);
```

3.7.4 Final details of parallel odd-even sort

Recall that we had developed the following parallel odd-even transposition sort algorithm:

```
Sort local keys;
for (phase = 0; phase < comm_sz; phase++) {
   partner = Compute_partner(phase, my_rank);
   if (I'm not idle) {
      Send my keys to partner;
      Receive keys from partner;
      if (my_rank < partner)
         Keep smaller keys;
      else
         Keep larger keys;
   }
}
```

In light of our discussion of safety in MPI, it probably makes sense to implement the send and the receive with a single call to `MPI_Sendrecv`:

```
MPI_Sendrecv(my_keys, n/comm_sz, MPI_INT, partner, 0,
      recv_keys, n/comm_sz, MPI_INT, partner, 0, comm,
      MPI_Status_ignore);
```

It only remains to identify which keys we keep. Suppose for the moment that we want to keep the smaller keys. Then we want to keep the smallest n/p keys in a collection of $2n/p$ keys. An obvious approach to doing this is to sort (using a serial sorting algorithm) the list of $2n/p$ keys and keep the first half of the list. However, sorting is a relatively expensive operation, and we can exploit the fact that we already have two sorted lists of n/p keys to reduce the cost by *merging* the two lists into a single list. In fact, we can do even better, because we don't need a fully general merge: once we've found the smallest n/p keys, we can quit. See Program 3.16.

To get the largest n/p keys, we simply reverse the order of the merge, that is, start with `local_n-1` and work backwards through the arrays. A final improvement avoids copying the arrays and simply swaps pointers (see Exercise 3.28).

Run-times for the version of parallel odd-even sort with the "final improvement" are shown in Table 3.9. Note that if parallel odd-even sort is run on a single processor, it will use whatever serial sorting algorithm we use to sort the local keys, so the times for a single process use serial quicksort, not serial odd-even sort, which would be *much* slower. We'll take a closer look at these times in Exercise 3.27.

Table 3.9 Run-Times of Parallel Odd-Even Sort (times are in milliseconds)

	Number of Keys (in thousands)				
Processes	*200*	*400*	*800*	*1600*	*3200*
1	88	190	390	830	1800
2	43	91	190	410	860
4	22	46	96	200	430
8	12	24	51	110	220
16	7.5	14	29	60	130

```
void Merge_low(
      int  my_keys[],     /* in/out    */
      int  recv_keys[],   /* in        */
      int  temp_keys[],   /* scratch   */
      int  local_n        /* = n/p, in */) {
   int m_i, r_i, t_i;

   m_i = r_i = t_i = 0;
   while (t_i < local_n) {
      if (my_keys[m_i] <= recv_keys[r_i]) {
         temp_keys[t_i] = my_keys[m_i];
         t_i++; m_i++;
      } else {
         temp_keys[t_i] = recv_keys[r_i];
         t_i++; r_i++;
      }
   }

   for (m_i = 0; m_i < local_n; m_i++)
      my_keys[m_i] = temp_keys[m_i];
}  /* Merge_low */
```

Program 3.16: The `Merge_low` function in parallel odd-even transposition sort

3.8 SUMMARY

MPI, or the Message-Passing Interface, is a library of functions that can be called from C, C++, or Fortran programs. Many systems use `mpicc` to compile MPI programs and `mpiexec` to run them. C MPI programs should include the `mpi.h` header file to get function prototypes and macros defined by MPI.

`MPI_Init` does the setup needed to run MPI. It should be called before other MPI functions are called. When your program doesn't use `argc` and `argv`, `NULL` can be passed for both arguments.

In MPI a **communicator** is a collection of processes that can send messages to each other. After an MPI program is started, MPI always creates a communicator consisting of all the processes. It's called `MPI_COMM_WORLD`.

Many parallel programs use the **single program, multiple data,** or **SPMD**, approach, whereby running a single program obtains the effect of running multiple different programs by including branches on data such as the process rank. When you're done using MPI, you should call `MPI_Finalize`.

To send a message from one MPI process to another, you can use `MPI_Send`. To receive a message, you can use `MPI_Recv`. The arguments to `MPI_Send` describe the contents of the message and its destination. The arguments to `MPI_Recv` describe the storage that the message can be received into, and where the message should be received from. `MPI_Recv` is **blocking**, that is, a call to `MPI_Recv` won't return until the

message has been received (or an error has occurred). The behavior of `MPI_Send` is defined by the MPI implementation. It can either block, or it can **buffer** the message. When it blocks, it won't return until the matching receive has started. If the message is buffered, MPI will copy the message into its own private storage, and `MPI_Send` will return as soon as the message is copied.

When you're writing MPI programs, it's important to differentiate between **local** and **global** variables. Local variables have values that are specific to the process on which they're defined, while global variables are the same on all the processes. In the trapezoidal rule program, the total number of trapezoids n was a global variable, while the left and right endpoints of each process' interval were local variables.

Most serial programs are **deterministic**, meaning if we run the same program with the same input we'll get the same output. Recall that parallel programs often don't possess this property—if multiple processes are operating more or less independently, the processes may reach various points at different times, depending on events outside the control of the process. Thus, parallel programs can be **nondeterministic**, that is, the same input can result in different outputs. If all the processes in an MPI program are printing output, the order in which the output appears may be different each time the program is run. For this reason, it's common in MPI programs to have a single process (e.g., process 0) handle all the output. This rule of thumb is usually ignored during debugging, when we allow each process to print debug information.

Most MPI implementations allow all the processes to print to `stdout` and `stderr`. However, every implementation we've encountered only allows at most one process (usually process 0 in `MPI_COMM_WORLD`) to read from `stdin`.

Collective communications involve all the processes in a communicator, so they're different from `MPI_Send` and `MPI_Recv`, which only involve two processes. To distinguish between the two types of communications, functions such as `MPI_Send` and `MPI_Recv` are often called **point-to-point** communications.

Two of the most commonly used collective communication functions are `MPI_Reduce` and `MPI_Allreduce`. `MPI_Reduce` stores the result of a global operation (e.g., a global sum) on a single designated process, while `MPI_Allreduce` stores the result on all the processes in the communicator.

In MPI functions such as `MPI_Reduce`, it may be tempting to pass the same actual argument to both the input and output buffers. This is called **argument aliasing**, and MPI explicitly prohibits aliasing an output argument with another argument.

We learned about a number of other important MPI collective communications:

- `MPI_Bcast` sends data from a single process to all the processes in a communicator. This is very useful if, for example, process 0 reads data from `stdin` and the data needs to be sent to all the processes.
- `MPI_Scatter` distributes the elements of an array among the processes. If the array to be distributed contains n elements, and there are p processes, then the first n/p are sent to process 0, the next n/p to process 1, and so on.
- `MPI_Gather` is the "inverse operation" to `MPI_Scatter`. If each process stores a subarray containing m elements, `MPI_Gather` will collect all of the elements onto

a designated process, putting the elements from process 0 first, then the elements from process 1, and so on.
- `MPI_Allgather` is like `MPI_Gather` except that it collects all of the elements onto *all* the processes.
- `MPI_Barrier` approximately synchronizes the processes; no process can return from a call to `MPI_Barrier` until all the processes in the communicator have started the call.

In distributed-memory systems there is no globally shared-memory, so partitioning global data structures among the processes is a key issue in writing MPI programs. For ordinary vectors and arrays, we saw that we could use block partitioning, cyclic partitioning, or block-cyclic partitioning. If the global vector or array has n components and there are p processes, a **block partition** assigns the first n/p to process 0, the next n/p to process 1, and so on. A **cyclic partition** assigns the elements in a "round-robin" fashion: the first element goes to 0, the next to 1, ..., the pth to $p-1$. After assigning the first p elements, we return to process 0, so the $(p+1)$st goes to process 0, the $(p+2)$nd to process 1, and so on. A **block-cyclic partition** assigns blocks of elements to the processes in a cyclic fashion.

Compared to operations involving only the CPU and main memory, sending messages is expensive. Furthermore, sending a given volume of data in fewer messages is usually less expensive than sending the same volume in more messages. Thus, it often makes sense to reduce the number of messages sent by combining the contents of multiple messages into a single message. MPI provides three methods for doing this: the `count` argument to communication functions, derived datatypes, and `MPI_Pack/Unpack`. Derived datatypes describe arbitrary collections of data by specifying the types of the data items and their relative positions in memory. In this chapter we took a brief look at the use of `MPI_Type_create_struct` to build a derived datatype. In the exercises, we'll explore some other methods, and we'll take a look at `MPI_Pack/Unpack`

When we time parallel programs, we're usually interested in elapsed time or "wall clock time," which is the total time taken by a block of code. It includes time in user code, time in library functions, time in operating system functions started by the user code, and idle time. We learned about two methods for finding wall clock time: `GET_TIME` and `MPI_Wtime`. `GET_TIME` is a macro defined in the file `timer.h` that can be downloaded from the book's website. It can be used in serial code as follows:

```
#include "timer.h"  // From the book's website
. . .
double start, finish, elapsed;
. . .
GET_TIME(start);
/* Code to be timed */
. . .
GET_TIME(finish);
elapsed = finish - start;
printf("Elapsed time = %e seconds\n", elapsed);
```

MPI provides a function, MPI_Wtime, that can be used instead of GET_TIME. In spite of this, timing parallel code is more complex, since ideally we'd like to synchronize the processes at the start of the code, and then report the time it took for the "slowest" process to complete the code. MPI_Barrier does a fairly good job of synchronizing the processes. A process that calls it will block until all the processes in the communicator have called it. We can use the following template for finding the run-time of MPI code:

```
double start, finish, loc_elapsed, elapsed;
. . .
MPI_Barrier(comm);
start = MPI_Wtime();
/* Code to be timed */
. . .
finish = MPI_Wtime();
loc_elapsed = finish - start;
MPI_Reduce(&loc_elapsed, &elapsed, 1, MPI_DOUBLE, MPI_MAX,
      0, comm);
if (my_rank == 0)
   printf("Elapsed time = %e seconds\n", elapsed);
```

A further problem with taking timings lies in the fact that there is ordinarily considerable variation if the same code is timed repeatedly. For example, the operating system may idle one or more of our processes so that other processes can run. Therefore, we typically take several timings and report their minimum.

After taking timings, we can use the **speedup** or the **efficiency** to evaluate the program performance. The speedup is the ratio of the serial run-time to the parallel run-time, and the efficiency is the speedup divided by the number of parallel processes. The ideal value for speedup is p, the number of processes, and the ideal value for the efficiency is 1. We rarely achieve these ideals, but it's not uncommon to see programs that get close to these values, especially when p is small and n, the problem size, is large. **Parallel overhead** is the part of the parallel run-time that's due to any additional work that isn't done by the serial program. In MPI programs, parallel overhead will come from communication. When p is large and n is small, it's not unusual for parallel overhead to dominate the total run-time and speedups and efficiencies can be quite low. If it's possible to increase the problem size (n) so that the efficiency doesn't decrease as p is increased, a parallel program is said to be **scalable**.

Recall that MPI_Send can either block or buffer its input. An MPI program is **unsafe** if its correct behavior depends on the fact that MPI_Send is buffering its input. This typically happens when multiple processes first call MPI_Send and then call MPI_Recv. If the calls to MPI_Send don't buffer the messages, then they'll block until the matching calls to MPI_Recv have started. However, this will never happen. For example, if both process 0 and process 1 want to send data to each other, and both send first and then receive, process 0 will wait forever for process 1 to call MPI_Recv, since process 1 is blocked in MPI_Send, and process 1 will wait forever for process 0.

That is, the processes will hang or **deadlock**—they'll block forever waiting for events that will never happen.

An MPI program can be checked for safety by replacing each call to `MPI_Send` with a call to `MPI_Ssend`. `MPI_Ssend` takes the same arguments as `MPI_Send`, but it always blocks until the matching receive has started. The extra "s" stands for *synchronous*. If the program completes correctly with `MPI_Ssend` for the desired inputs and communicator sizes, then the program is safe.

An unsafe MPI program can be made safe in several ways. The programmer can schedule the calls to `MPI_Send` and `MPI_Recv` so that some processes (e.g., even-ranked processes) first call `MPI_Send` while others (e.g., odd-ranked processes) first call `MPI_Recv`. Alternatively, we can use `MPI_Sendrecv` or `MPI_Sendrecv_replace`. These functions execute both a send and a receive, but they're guaranteed to schedule them so that the program won't hang or crash. `MPI_Sendrecv` uses different arguments for the send and the receive buffers, while `MPI_Sendrecv_replace` uses the same buffer for both.

3.9 EXERCISES

3.1. What happens in the greetings program if, instead of `strlen(greeting)+1`, we use `strlen(greeting)` for the length of the message being sent by processes $1, 2, \ldots,$ `comm_sz`-1? What happens if we use `MAX_STRING` instead of `strlen(greeting) + 1`? Can you explain these results?

3.2. Modify the trapezoidal rule so that it will correctly estimate the integral even if `comm_sz` doesn't evenly divide n. (You can still assume that $n \geq$ `comm_sz`.)

3.3. Determine which of the variables in the trapezoidal rule program are local and which are global.

3.4. Modify the program that just prints a line of output from each process (`mpi_output.c`) so that the output is printed in process rank order: process 0s output first, then process 1s, and so on.

3.5. In a binary tree, there is a unique shortest path from each node to the root. The length of this path is often called the **depth** of the node. A binary tree in which every nonleaf has two children is called a **full** binary tree, and a full binary tree in which every leaf has the same depth is sometimes called a **complete** binary tree. See Figure 3.14. Use the principle of mathematical induction to prove that if T is a complete binary tree with n leaves, then the depth of the leaves is $\log_2(n)$.

3.6. Suppose `comm_sz` $= 4$ and suppose that $\mathbf{x}$ is a vector with $n = 14$ components.

a. How would the components of $\mathbf{x}$ be distributed among the processes in a program that used a block distribution?

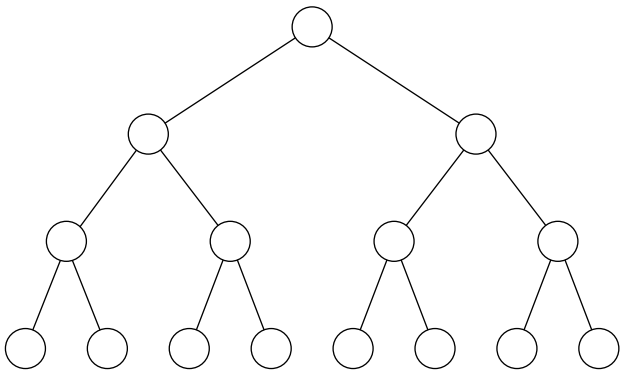


FIGURE 3.14

A complete binary tree

b. How would the components of **x** be distributed among the processes in a program that used a cyclic distribution?

c. How would the components of **x** be distributed among the processes in a program that used a block-cyclic distribution with blocksize $b = 2$?

You should try to make your distributions general so that they could be used regardless of what `comm_sz` and n are. You should also try to make your distributions "fair" so that if q and r are any two processes, the difference between the number of components assigned to q and the number of components assigned to r is as small as possible.

3.7. What do the various MPI collective functions do if the communicator contains a single process?

3.8. Suppose `comm_sz` $= 8$ and $n = 16$.

a. Draw a diagram that shows how `MPI_Scatter` can be implemented using tree-structured communication with `comm_sz` processes when process 0 needs to distribute an array containing n elements.

b. Draw a diagram that shows how `MPI_Gather` can be implemented using tree-structured communication when an n-element array that has been distributed among `comm_sz` processes needs to be gathered onto process 0.

3.9. Write an MPI program that implements multiplication of a vector by a scalar and dot product. The user should enter two vectors and a scalar, all of which are read in by process 0 and distributed among the processes. The results are calculated and collected onto process 0, which prints them. You can assume that n, the order of the vectors, is evenly divisible by `comm_sz`.

3.10. In the `Read_vector` function shown in Program 3.9, we use `local_n` as the actual argument for two of the formal arguments to `MPI_Scatter`: `send_count` and `recv_count`. Why is it OK to alias these arguments?

3.11. Finding **prefix sums** is a generalization of global sum. Rather than simply finding the sum of n values,

$$x_0 + x_1 + \cdots + x_{n-1},$$

the prefix sums are the n partial sums

$$x_0,\ x_0 + x_1,\ x_0 + x_1 + x_2,\ \ldots,\ x_0 + x_1 + \cdots + x_{n-1}.$$

a. Devise a serial algorithm for computing the n prefix sums of an array with n elements.

b. Parallelize your serial algorithm for a system with n processes, each of which is storing one of the `x_i`s.

c. Suppose $n = 2^k$ for some positive integer k. Can you devise a serial algorithm and a parallelization of the serial algorithm so that the parallel algorithm requires only k communication phases?

d. MPI provides a collective communication function, `MPI_Scan`, that can be used to compute prefix sums:

```
int MPI_Scan(
      void*         sendbuf_p   /* in  */,
      void*         recvbuf_p   /* out */,
      int           count       /* in  */,
      MPI_Datatype  datatype    /* in  */,
      MPI_Op        op          /* in  */,
      MPI_Comm      comm        /* in  */);
```

It operates on arrays with `count` elements; both `sendbuf_p` and `recvbuf_p` should refer to blocks of `count` elements of type `datatype`. The `op` argument is the same as `op` for `MPI_Reduce`. Write an MPI program that generates a random array of `count` elements on each MPI process, finds the prefix sums, and prints the results.

3.12. An alternative to a butterfly-structured allreduce is a **ring-pass** structure. In a ring-pass, if there are p processes, each process q sends data to process $q + 1$, except that process $p - 1$ sends data to process 0. This is repeated until each process has the desired result. Thus, we can implement allreduce with the following code:

```
sum = temp_val = my_val;
for (i = 1; i < p; i++) {
   MPI_Sendrecv_replace(&temp_val, 1, MPI_INT, dest,
         sendtag, source, recvtag, comm, &status);
   sum += temp_val;
}
```

a. Write an MPI program that implements this algorithm for allreduce. How does its performance compare to the butterfly-structured allreduce?

b. Modify the MPI program you wrote in the first part so that it implements prefix sums.

3.13. `MPI_Scatter` and `MPI_Gather` have the limitation that each process must send or receive the same number of data items. When this is not the case, we must use the MPI functions `MPI_Gatherv` and `MPI_Scatterv`. Look at the man pages for these functions, and modify your vector sum, dot product program so that it can correctly handle the case when n isn't evenly divisible by `comm_sz`.

3.14. **a.** Write a serial C program that defines a two-dimensional array in the main function. Just use numeric constants for the dimensions:

```
int two_d[3][4];
```

Initialize the array in the main function. After the array is initialized, call a function that attempts to print the array. The prototype for the function should look something like this.

```
void Print_two_d(int two_d[][], int rows, int cols);
```

After writing the function try to compile the program. Can you explain why it won't compile?

b. After consulting a C reference (e.g., Kernighan and Ritchie [29]), modify the program so that it will compile and run, but so that it still uses a two-dimensional C array.

3.15. What is the relationship between the "row-major" storage for two-dimensional arrays that we discussed in Section 2.2.3 and the one-dimensional storage we use in Section 3.4.9?

3.16. Suppose `comm_sz` $= 8$ and the vector $\mathbf{x} = (0, 1, 2, \ldots, 15)$ has been distributed among the processes using a block distribution. Draw a diagram illustrating the steps in a butterfly implementation of allgather of $\mathbf{x}$.

3.17. `MPI_Type_contiguous` can be used to build a derived datatype from a collection of contiguous elements in an array. Its syntax is

```
int MPI_Type_contiguous(
      int           count        /* in  */,
      MPI_Datatype  old_mpi_t    /* in  */,
      MPI_Datatype* new_mpi_t_p  /* out */);
```

Modify the `Read_vector` and `Print_vector` functions so that they use an MPI datatype created by a call to `MPI_Type_contiguous` and a `count` argument of 1 in the calls to `MPI_Scatter` and `MPI_Gather`.

3.18. `MPI_Type_vector` can be used to build a derived datatype from a collection of blocks of elements in an array as long as the blocks all have the same size and they're equally spaced. Its syntax is

```
int MPI_Type_vector(
      int           count        /* in  */,
      int           blocklength  /* in  */,
```

```
int            stride        /* in  */,
MPI_Datatype   old_mpi_t     /* in  */,
MPI_Datatype*  new_mpi_t_p   /* out */);
```

For example, if we had an array `x` of 18 **double**s and we wanted to build a type corresponding to the elements in positions 0, 1, 6, 7, 12, 13, we could call

```
int MPI_Type_vector(3, 2, 6, MPI_DOUBLE, &vect_mpi_t);
```

since the type consists of 3 blocks, each of which has 2 elements, and the spacing between the starts of the blocks is 6 **double**s.

Write `Read_vector` and `Print_vector` functions that will allow process 0 to read and print, respectively, a vector with a block-cyclic distribution. But beware! Do *not* use `MPI_Scatter` or `MPI_Gather`. There is a technical issue involved in using these functions with types created with `MPI_Type_vector`. (See, for example, [23].) Just use a loop of sends on process 0 in `Read_vector` and a loop of receives on process 0 in `Print_vector`. The other processes should be able to complete their calls to `Read_vector` and `Print_vector` with a single call to `MPI_Recv` and `MPI_Send`. The communication on process 0 should use a derived datatype created by `MPI_Type_vector`. The calls on the other processes should just use the `count` argument to the communication function, since they're receiving/sending elements that they will store in contiguous array locations.

3.19. `MPI_Type_indexed` can be used to build a derived datatype from arbitrary array elements. Its syntax is

```
int MPI_Type_indexed(
      int            count                      /* in  */,
      int            array_of_blocklengths[]    /* in  */,
      int            array_of_displacements[]   /* in  */,
      MPI_Datatype   old_mpi_t                  /* in  */,
      MPI_Datatype*  new_mpi_t_p)               /* out */);
```

Unlike `MPI_Type_create_struct`, the displacements are measured in units of `old_mpi_t`—not bytes. Use `MPI_Type_indexed` to create a derived datatype that corresponds to the upper triangular part of a square matrix. For example, in the 4×4 matrix

$$\begin{pmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \\ 12 & 13 & 14 & 15 \end{pmatrix}$$

the upper triangular part is the elements $0, 1, 2, 3, 5, 6, 7, 10, 11, 15$. Process 0 should read in an $n \times n$ matrix as a one-dimensional array, create the derived datatype, and send the upper triangular part with a single call to `MPI_Send`. Process 1 should receive the upper triangular part with a single call to `MPI_Recv` and then print the data it received.

3.20. The functions `MPI_Pack` and `MPI_Unpack` provide an alternative to derived datatypes for grouping data. `MPI_Pack` copies the data to be sent, one block at a time, into a user-provided buffer. The buffer can then be sent and received. After the data is received, `MPI_Unpack` can be used to unpack it from the receive buffer. The syntax of `MPI_Pack` is

```
int MPI_Pack(
      void*          in_buf          /* in     */,
      int            in_buf_count    /* in     */,
      MPI_Datatype   datatype        /* in     */,
      void*          pack_buf        /* out    */,
      int            pack_buf_sz     /* in     */,
      int*           position_p      /* in/out */,
      MPI_Comm       comm            /* in     */);
```

We could therefore pack the input data to the trapezoidal rule program with the following code:

```
char pack_buf[100];
int position = 0;

MPI_Pack(&a, 1, MPI_DOUBLE, pack_buf, 100, &position, comm);
MPI_Pack(&b, 1, MPI_DOUBLE, pack_buf, 100, &position, comm);
MPI_Pack(&n, 1, MPI_INT, pack_buf, 100, &position, comm);
```

The key is the `position` argument. When `MPI_Pack` is called, position should refer to the first available slot in `pack_buf`. When `MPI_Pack` returns, it refers to the first available slot *after* the data that was just packed, so after process 0 executes this code, all the processes can call `MPI_Bcast`:

```
MPI_Bcast(pack_buf, 100, MPI_PACKED, 0, comm);
```

Note that the MPI datatype for a packed buffer is `MPI_PACKED`. Now the other processes can unpack the data using: `MPI_Unpack`:

```
int MPI_Unpack(
      void*          pack_buf        /* in     */,
      int            pack_buf_sz     /* in     */,
      int*           position_p      /* in/out */,
      void*          out_buf         /* out    */,
      int            out_buf_count   /* in     */,
      MPI_Datatype   datatype        /* in     */,
      MPI_Comm       comm            /* in     */);
```

This can be used by "reversing" the steps in `MPI_Pack`, that is, the data is unpacked one block at a time starting with `position = 0`.

Write another `Get_input` function for the trapezoidal rule program. This one should use `MPI_Pack` on process 0 and `MPI_Unpack` on the other processes.

3.21. How does your system compare to ours? What run-times does your system get for matrix-vector multiplication? What kind of variability do you see in

the times for a given value of `comm_sz` and n? Do the results tend to cluster around the minimum, the mean, or the median?

3.22. Time our implementation of the trapezoidal rule that uses `MPI_Reduce`. How will you choose n, the number of trapezoids? How do the minimum times compare to the mean and median times? What are the speedups? What are the efficiencies? On the basis of the data you collected, would you say that the trapezoidal rule is scalable?

3.23. Although we don't know the internals of the implementation of `MPI_Reduce`, we might guess that it uses a structure similar to the binary tree we discussed. If this is the case, we would expect that its run-time would grow roughly at the rate of $\log_2(p)$, since there are roughly $\log_2(p)$ levels in the tree. (Here, $p =$ `comm_sz`.) Since the run-time of the serial trapezoidal rule is roughly proportional to n, the number of trapezoids, and the parallel trapezoidal rule simply applies the serial rule to n/p trapezoids on each process, with our assumption about `MPI_Reduce`, we get a formula for the overall run-time of the parallel trapezoidal rule that looks like

$$T_{\text{parallel}}(n,p) \approx a \times \frac{n}{p} + b\log_2(p)$$

for some constants a and b.

- **a.** Use the formula, the times you've taken in Exercise 3.22, and your favorite program for doing mathematical calculations (e.g., MATLAB®) to get a least-squares estimate of the values of a and b.
- **b.** Comment on the quality of the predicted run-times using the formula and the values for a and b computed in part (a).

3.24. Take a look at Programming Assignment 3.7. The code that we outlined for timing the cost of sending messages should work even if the `count` argument is zero. What happens on your system when the `count` argument is 0? Can you explain why you get a nonzero elapsed time when you send a zero-byte message?

3.25. If `comm_sz` $= p$, we mentioned that the "ideal" speedup is p. Is it possible to do better?

- **a.** Consider a parallel program that computes a vector sum. If we only time the vector sum—that is, we ignore input and output of the vectors—how might this program achieve speedup greater than p?
- **b.** A program that achieves speedup greater than p is said to have **superlinear** speedup. Our vector sum example only achieved superlinear speedup by overcoming certain "resource limitations." What were these resource limitations? Is it possible for a program to obtain superlinear speedup without overcoming resource limitations?

3.26. Serial odd-even transposition sort of an n-element list can sort the list in considerably fewer than n phases. As an extreme example, if the input list is already sorted, the algorithm requires 0 phases.
- **a.** Write a serial `Is_sorted` function that determines whether a list is sorted.
- **b.** Modify the serial odd-even transposition sort program so that it checks whether the list is sorted after each phase.
- **c.** If this program is tested on a random collection of n-element lists, roughly what fraction get improved performance by checking whether the list is sorted?

3.27. Find the speedups and efficiencies of the parallel odd-even sort. Does the program obtain linear speedups? Is it scalable? Is it strongly scalable? Is it weakly scalable?

3.28. Modify the parallel odd-even transposition sort so that the `Merge` functions simply swap array pointers after finding the smallest or largest elements. What effect does this change have on the overall run-time?

3.10 PROGRAMMING ASSIGNMENTS

3.1. Use MPI to implement the histogram program discussed in Section 2.7.1. Have process 0 read in the input data and distribute it among the processes. Also have process 0 print out the histogram.

3.2. Suppose we toss darts randomly at a square dartboard, whose bullseye is at the origin, and whose sides are 2 feet in length. Suppose also that there's a circle inscribed in the square dartboard. The radius of the circle is 1 foot, and it's area is π square feet. If the points that are hit by the darts are uniformly distributed (and we always hit the square), then the number of darts that hit inside the circle should approximately satisfy the equation

$$\frac{\text{number in circle}}{\text{total number of tosses}} = \frac{\pi}{4},$$

since the ratio of the area of the circle to the area of the square is $\pi/4$.

We can use this formula to estimate the value of π with a random number generator:

```
number_in_circle = 0;
for (toss = 0; toss < number_of_tosses; toss++) {
   x = random double between -1 and 1;
   y = random double between -1 and 1;
   distance_squared = x*x + y*y;
   if (distance_squared <= 1) number_in_circle++;
}
pi_estimate = 4*number_in_circle/((double) number_of_tosses);
```

This is called a "Monte Carlo" method, since it uses randomness (the dart tosses).

Write an MPI program that uses a Monte Carlo method to estimate π. Process 0 should read in the total number of tosses and broadcast it to the other processes. Use `MPI_Reduce` to find the global sum of the local variable `number_in_circle`, and have process 0 print the result. You may want to use **long long int**s for the number of hits in the circle and the number of tosses, since both may have to be very large to get a reasonable estimate of π.

3.3. Write an MPI program that computes a tree-structured global sum. First write your program for the special case in which `comm_sz` is a power of two. Then, after you've gotten this version working, modify your program so that it can handle any `comm_sz`.

3.4. Write an MPI program that computes a global sum using a butterfly. First write your program for the special case in which `comm_sz` is a power of two. Can you modify your program so that it will handle any number of processes?

3.5. Implement matrix-vector multiplication using a block-column distribution of the matrix. You can have process 0 read in the matrix and simply use a loop of sends to distribute it among the processes. Assume the matrix is square of order n and that n is evenly divisible by `comm_sz`. You may want to look at the MPI function `MPI_Reduce_scatter`.

3.6. Implement matrix-vector multiplication using a block-submatrix distribution of the matrix. Assume that the vectors are distributed among the diagonal processes. Once again, you can have process 0 read in the matrix and aggregate the sub-matrices before sending them to the processes. Assume `comm_sz` is a perfect square and that $\sqrt{\texttt{comm\_sz}}$ evenly divides the order of the matrix.

3.7. A **ping-pong** is a communication in which two messages are sent, first from process A to process B (ping) and then from process B back to process A (pong). Timing blocks of repeated ping-pongs is a common way to estimate the cost of sending messages. Time a ping-pong program using the C `clock` function on your system. How long does the code have to run before `clock` gives a nonzero run-time? How do the times you got with the `clock` function compare to times taken with `MPI_Wtime`?

3.8. Parallel merge sort starts with $n/$`comm_sz` keys assigned to each process. It ends with all the keys stored on process 0 in sorted order. To achieve this, it uses the same tree-structured communication that we used to implement a global sum. However, when a process receives another process' keys, it merges the new keys into its already sorted list of keys. Write a program that implements parallel mergesort. Process 0 should read in n and broadcast it to the other processes. Each process should use a random number generator to create a local list of `n/comm_sz` `int`s. Each process should then sort its local list, and

process 0 should gather and print the local lists. Then the processes should use tree-structured communication to merge the global list onto process 0, which prints the result.

3.9. Write a program that can be used to determine the cost of changing the distribution of a distributed data structure. How long does it take to change from a block distribution of a vector to a cyclic distribution? How long does the reverse redistribution take?

This page intentionally left blank

CHAPTER

Shared-Memory Programming with Pthreads

4

Recall that from a programmer's point of view a shared-memory system is one in which all the cores can access all the memory locations (see Figure 4.1). Thus, an obvious approach to the problem of coordinating the work of the cores is to specify that certain memory locations are "shared." This is a very natural approach to parallel programming. Indeed, we might well wonder why all parallel programs don't use this shared-memory approach. However, we'll see in this chapter that there are problems in programming shared-memory systems, problems that are often different from the problems encountered in distributed-memory programming.

For example, in Chapter 2 we saw that if different cores attempt to update a single shared-memory location, then the contents of the shared location can be unpredictable. The code that updates the shared location is an example of a *critical section.* We'll see some other examples of critical sections, and we'll learn several methods for controlling access to a critical section.

We'll also learn about other issues and techniques in shared-memory programming. In shared-memory programming, an instance of a program running on a processor is usually called a **thread** (unlike MPI, where it's called a process). We'll learn how to synchronize threads so that each thread will wait to execute a block of statements until another thread has completed some work. We'll learn how to put a thread "to sleep" until a condition has occurred. We'll see that there are some circumstances in which it may at first seem that a critical section must be quite large. However, we'll also see that there are tools that sometimes allow us to "fine-tune" access to these large blocks of code so that more of the program can truly be executed in parallel. We'll see that the use of cache memories can actually cause a shared-memory program to run more slowly. Finally, we'll see that functions that "maintain state" between successive calls can cause inconsistent or even incorrect results.

In this chapter we'll be using POSIX® threads for most of our shared-memory functions. In the next chapter we'll look at an alternative approach to shared-memory programming called OpenMP.

4.1 PROCESSES, THREADS, AND PTHREADS

Recall from Chapter 2 that in shared-memory programming, a thread is somewhat analogous to a process in MPI programming. However, it can, in principle, be

An Introduction to Parallel Programming. DOI: 10.1016/B978-0-12-374260-5.00004-X

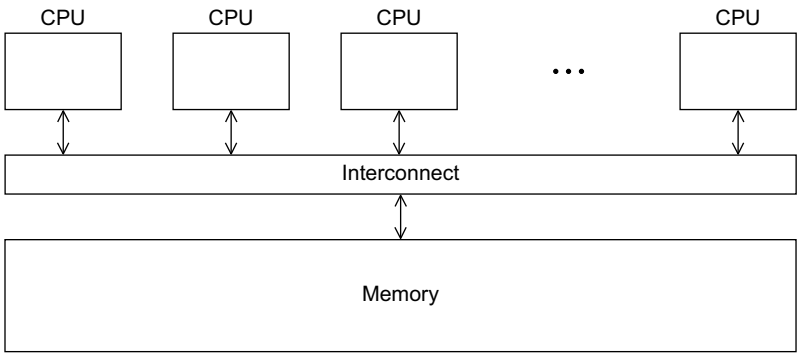


FIGURE 4.1

A shared-memory system

"lighter-weight." A process is an instance of a running (or suspended) program. In addition to its executable, it consists of the following:

- A block of memory for the stack
- A block of memory for the heap
- Descriptors of resources that the system has allocated for the process—for example, file descriptors
- Security information—for example, information about which hardware and software resources the process can access
- Information about the state of the process, such as whether the process is ready to run or is waiting on a resource, the content of the registers including the program counter, and so on

In most systems, by default, a process' memory blocks are private: another process can't directly access the memory of a process unless the operating system intervenes. This makes sense. If you're using a text editor to write a program (one process—the running text editor), you don't want your browser (another process) overwriting your text editor's memory. This is even more crucial in a multiuser environment. One user's processes shouldn't be allowed access to the memory of another user's processes.

However, this isn't what we want when we're running shared-memory programs. At a minimum, we'd like certain variables to be available to multiple processes, so shared-memory "processes" typically allow much easier access to each others' memory. They also often share things such as access to `stdout`. In fact, it's conceivable that they share pretty much everything that's process specific, except their stacks and their program counters. This can be relatively easily arranged by starting a single process and then having the process start these "lighter-weight" processes. For this reason, they're often called **light-weight processes**.

The more commonly used term, **thread**, comes from the concept of "thread of control." A thread of control is just a sequence of statements in a program. The term

suggests a stream of control in a single process, and in a shared-memory program a single *process* may have multiple *threads* of control.

As we noted earlier, in this chapter the particular implementation of threads that we'll be using is called POSIX threads or, more often, **Pthreads**. POSIX [41] is a standard for Unix-like operating systems—for example, Linux and Mac OS X. It specifies a variety of facilities that should be available in such systems. In particular, it specifies an application programming interface (API) for *multithreaded* programming.

Pthreads is not a programming language (such as C or Java). Rather, like MPI, Pthreads specifies a *library* that can be linked with C programs. Unlike MPI, the Pthreads API is only available on POSIX systems—Linux, Mac OS X, Solaris, HPUX, and so on. Also unlike MPI, there are a number of other widely used specifications for multithreaded programming: Java threads, Windows threads, Solaris threads. However, all of the thread specifications support the same basic ideas, so once you've learned how to program in Pthreads, it won't be difficult to learn how to program another thread API.

Since Pthreads is a C library, it can, in principle, be used in C++ programs. However, there is an effort underway to develop a C++ standard (C++0x) for shared-memory programming. It may make sense to use it rather than Pthreads if you're writing C++ programs.

4.2 HELLO, WORLD

Let's get started. Let's take a look at a Pthreads program. Program 4.1 shows a program in which the main function starts up several threads. Each thread prints a message and then quits.

4.2.1 Execution

The program is compiled like an ordinary C program, with the possible exception that we may need to link in the Pthreads library:[1]

```
$ gcc -g -Wall -o pth_hello pth_hello.c -lpthread
```

The `-lpthread` tells the compiler that we want to link in the Pthreads library. Note that it's `-lpthread`, *not* `-lpthreads`. On some systems the compiler will automatically link in the library, and `-lpthread` won't be needed.

To run the program, we just type

```
$ ./pth_hello <number of threads>
```

[1] Recall that the dollar sign (`$`) is the shell prompt, so it shouldn't be typed in. Also recall that for the sake of explicitness, we assume that we're using the Gnu C compiler, `gcc`, and we always use the options `-g`, `-Wall`, and `-o`. See Section 2.9 for further information.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <pthread.h>
4
5  /* Global variable:  accessible to all threads */
6  int thread_count;
7
8  void* Hello(void* rank);  /* Thread function */
9
10 int main(int argc, char* argv[]) {
11    long       thread;  /* Use long in case of a 64-bit system */
12    pthread_t* thread_handles;
13
14    /* Get number of threads from command line */
15    thread_count = strtol(argv[1], NULL, 10);
16
17    thread_handles = malloc (thread_count*sizeof(pthread_t));
18
19    for (thread = 0; thread < thread_count; thread++)
20       pthread_create(&thread_handles[thread], NULL,
21           Hello, (void*) thread);
22
23    printf("Hello from the main thread\n");
24
25    for (thread = 0; thread < thread_count; thread++)
26       pthread_join(thread_handles[thread], NULL);
27
28    free(thread_handles);
29    return 0;
30 }  /* main */
31
32 void* Hello(void* rank) {
33    long my_rank = (long) rank
          /* Use long in case of 64-bit system */
34
35    printf("Hello from thread %ld of %d\n", my_rank,
          thread_count);
36
37    return NULL;
38 }  /* Hello */
```

Program 4.1: A Pthreads "hello, world" program

For example, to run the program with one thread, we type

```
$ ./pth_hello 1
```

and the output will look something like this:

```
Hello from the main thread
Hello from thread 0 of 1
```

To run the program with four threads, we type

```
$ ./pth_hello 4
```

and the output will look something like this:

```
Hello from the main thread
Hello from thread 0 of 4
Hello from thread 1 of 4
Hello from thread 2 of 4
Hello from thread 3 of 4
```

4.2.2 Preliminaries

Let's take a closer look at the source code in Program 4.1. First notice that this *is* just a C program with a `main` function and one other function. The program includes the familiar `stdio.h` and `stdlib.h` header files. However, there's a lot that's new and different. In Line 3 we include `pthread.h`, the Pthreads header file, which declares the various Pthreads functions, constants, types, and so on.

In Line 6 we define a *global* variable `thread_count`. In Pthreads programs, global variables are shared by all the threads. Local variables and function arguments—that is, variables declared in functions—are (ordinarily) private to the thread executing the function. If several threads are executing the same function, each thread will have its own private copies of the local variables and function arguments. This makes sense if you recall that each thread has its own stack.

We should keep in mind that global variables can introduce subtle and confusing bugs. For example, suppose we write a program in which we declare a global variable `int x`. Then we write a function `f` in which we intend to use a local variable called `x`, but we forget to declare it. The program will compile with no warnings, since `f` has access to the global `x`. But when we run the program, it produces very strange output, which we eventually determine to have been caused by the fact that the global variable `x` has a strange value. Days later, we finally discover that the strange value came from `f`. As a rule of thumb, we should try to limit our use of global variables to situations in which they're really needed—for example, for a shared variable.

In Line 15 the program gets the number of threads it should start from the command line. Unlike MPI programs, Pthreads programs are typically compiled and run just like serial programs, and one relatively simple way to specify the number of threads that should be started is to use a command-line argument. This isn't a requirement, it's simply a convenient convention we'll be using.

The `strtol` function converts a string into a `long int`. It's declared in `stdlib.h`, and its syntax is

```
long strtol(
          const char*   number_p   /* in  */,
          char**        end_p      /* out */,
          int           base       /* in  */);
```

It returns a `long int` corresponding to the string referred to by `number_p`. The base of the representation of the number is given by the `base` argument. If `end_p` isn't `NULL`, it will point to the first invalid (that is, nonnumeric) character in `number_p`.

4.2.3 Starting the threads

As we already noted, unlike MPI programs, in which the processes are usually started by a script, in Pthreads the threads are started by the program executable. This introduces a bit of additional complexity, as we need to include code in our program to explicitly start the threads, and we need data structures to store information on the threads.

In Line 17 we allocate storage for one `pthread_t` object for each thread. The `pthread_t` data structure is used for storing thread-specific information. It's declared in `pthread.h`.

The `pthread_t` objects are examples of **opaque** objects. The actual data that they store is system specific, and their data members aren't directly accessible to user code. However, the Pthreads standard guarantees that a `pthread_t` object does store enough information to uniquely identify the thread with which it's associated. So, for example, there is a Pthreads function that a thread can use to retrieve its associated `pthread_t` object, and there is a Pthreads function that can determine whether two threads are in fact the same by examining their associated `pthread_t` objects.

In Lines 19–21, we use the `pthread_create` function to start the threads. Like most Pthreads functions, its name starts with the string `pthread_`. The syntax of `pthread_create` is

```
int pthread_create(
      pthread_t*             thread_p                  /* out */,
      const pthread_attr_t*  attr_p                    /* in  */,
      void*                  (*start_routine)(void*)   /* in  */,
      void*                  arg_p                     /* in  */);
```

The first argument is a pointer to the appropriate `pthread_t` object. Note that the object is not allocated by the call to `pthread_create`; it must be allocated *before* the call. We won't be using the second argument, so we just pass the argument `NULL` in our function call. The third argument is the function that the thread is to run, and the last argument is a pointer to the argument that should be passed to the function `start_routine`. The return value for most Pthreads functions indicates if there's been an error in the function call. In order to reduce the clutter in our examples, in this chapter (as in most of the rest of the book) we'll generally ignore the return values of Pthreads functions.

Let's take a closer look at the last two arguments. The function that's started by `pthread_create` should have a prototype that looks something like this:

```
void* thread_function(void* args_p);
```

Recall that the type `void*` can be cast to any pointer type in C, so `args_p` can point to a list containing one or more values needed by `thread_function`. Similarly, the return value of `thread_function` can point to a list of one or more values. In our call to `pthread_create`, the final argument is a fairly common kluge: we're effectively assigning each thread a unique integer *rank*. Let's first look at why we are doing this; then we'll worry about the details of how to do it.

Consider the following problem: We start a Pthreads program that uses two threads, but one of the threads encounters an error. How do we, the users, know which thread encountered the error? We can't just print out the `pthread_t` object, since it's opaque. However, if when we start the threads, we assign the first thread rank 0, and the second thread rank 1, we can easily determine which thread ran into trouble by just including the thread's rank in the error message.

Since the thread function takes a `void*` argument, we could allocate one `int` in `main` for each thread and assign each allocated `int` a unique value. When we start a thread, we could then pass a pointer to the appropriate `int` in the call to `pthread_create`. However, most programmers resort to some trickery with casts. Instead of creating an `int` in `main` for the "rank," we cast the loop variable `thread` to have type `void*`. Then in the thread function, `hello`, we cast the argument back to a `long` (Line 33).

The result of carrying out these casts is "system-defined," but most C compilers do allow this. However, if the size of pointer types is different from the size of the integer type you use for the rank, you may get a warning. On the machines we used, pointers are 64 bits, and `int`s are only 32 bits, so we use `long` instead of `int`.

Note that our method of assigning thread ranks and, indeed, the thread ranks themselves are just a convenient convention that we'll use. There is no requirement that a thread rank be passed in the call to `pthread_create`. Indeed there's no requirement that a thread be assigned a rank.

Also note that there is no technical reason for each thread to run the same function; we could have one thread run `hello`, another run `goodbye`, and so on. However, as with the MPI programs, we'll typically use "single program, multiple data" style parallelism with our Pthreads programs. That is, each thread will run the same thread function, but we'll obtain the effect of different thread functions by branching within a thread.

4.2.4 Running the threads

The thread that's running the `main` function is sometimes called the **main thread**. Hence, after starting the threads, it prints the message

```
Hello from the main thread
```

In the meantime, the threads started by the calls to `pthread_create` are also running. They get their ranks by casting in Line 33, and then print their messages. Note that when a thread is done, since the type of its function has a return value, the thread should return something. In this example, the threads don't actually need to return anything, so they return `NULL`.

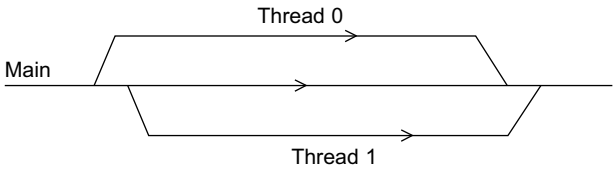


FIGURE 4.2

Main thread forks and joins two threads

In Pthreads, the programmer doesn't directly control where the threads are run.[2] There's no argument in `pthread_create` saying which core should run which thread. Thread placement is controlled by the operating system. Indeed, on a heavily loaded system, the threads may all be run on the same core. In fact, if a program starts more threads than cores, we should expect multiple threads to be run on a single core. However, if there is a core that isn't being used, operating systems will typically place a new thread on such a core.

4.2.5 Stopping the threads

In Lines 25 and 26, we call the function `pthread_join` once for each thread. A single call to `pthread_join` will wait for the thread associated with the `pthread_t` object to complete. The syntax of `pthread_join` is

```
int pthread_join(
      pthread_t  thread     /* in  */,
      void**     ret_val_p  /* out */);
```

The second argument can be used to receive any return value computed by the thread. So in our example, each thread executes a `return` and, eventually, the main thread will call `pthread_join` for that thread to complete the termination.

This function is called `pthread_join` because of a diagramming style that is often used to describe the threads in a multithreaded process. If we think of the main thread as a single line in our diagram, then, when we call `pthread_create`, we can create a *branch* or *fork* off the main thread. Multiple calls to `pthread_create` will result in multiple branches or forks. Then, when the threads started by `pthread_create` terminate, the diagram shows the branches *joining* the main thread. See Figure 4.2.

4.2.6 Error checking

In the interest of keeping the program compact and easy to read, we have resisted the temptation to include many details that would therefore be important in a "real" program. The most likely source of problems in this example (and in many programs) is the user input or lack of it. It would therefore be a very good idea to check that the program was started with command line arguments, and, if it was, to check the actual

[2]Some systems (for example, some implementations of Linux) do allow the programmer to specify where a thread is run. However, these constructions will not be portable.

value of the number of threads to see if it's reasonable. If you visit the book's website, you can download a version of the program that includes this basic error checking.

It may also be a good idea to check the error codes returned by the Pthreads functions. This can be especially useful when you're just starting to use Pthreads and some of the details of function use aren't completely clear.

4.2.7 Other approaches to thread startup

In our example, the user specifies the number of threads to start by typing in a command-line argument. The main thread then creates all of the "subsidiary" threads. While the threads are running, the main thread prints a message, and then waits for the other threads to terminate. This approach to threaded programming is very similar to our approach to MPI programming, in which the MPI system starts a collection of processes and waits for them to complete.

There is, however, a very different approach to the design of multithreaded programs. In this approach, subsidiary threads are only started as the need arises. As an example, imagine a Web server that handles requests for information about highway traffic in the San Francisco Bay Area. Suppose that the main thread receives the requests and subsidiary threads actually fulfill the requests. At 1 o'clock on a typical Tuesday morning, there will probably be very few requests, while at 5 o'clock on a typical Tuesday evening, there will probably be thousands of requests. Thus, a natural approach to the design of this Web server is to have the main thread start subsidiary threads when it receives requests.

Now, we do need to note that thread startup necessarily involves some overhead. The time required to start a thread will be much greater than, say, a floating point arithmetic operation, so in applications that need maximum performance the "start threads as needed" approach may not be ideal. In such a case, it may make sense to use a somewhat more complicated scheme—a scheme that has characteristics of both approaches. Our main thread can start all the threads it anticipates needing at the beginning of the program (as in our example program). However, when a thread has no work, instead of terminating, it can sit idle until more work is available. In Programming Assignment 4.5 we'll look at how we might implement such a scheme.

4.3 MATRIX-VECTOR MULTIPLICATION

Let's take a look at writing a Pthreads matrix-vector multiplication program. Recall that if $A = (a_{ij})$ is an $m \times n$ matrix and $\mathbf{x} = (x_0, x_1, \ldots, x_{n-1})^T$ is an n-dimensional column vector,[3] then the matrix-vector product $A\mathbf{x} = \mathbf{y}$ is an m-dimensional column vector, $\mathbf{y} = (y_0, y_1, \ldots, y_{m-1})^T$ in which the ith component y_i is obtained by finding

[3]Recall that we use the convention that matrix and vector subscripts start with 0. Also recall that if $\mathbf{b}$ is a matrix or a vector, then $\mathbf{b}^T$ denotes its transpose.

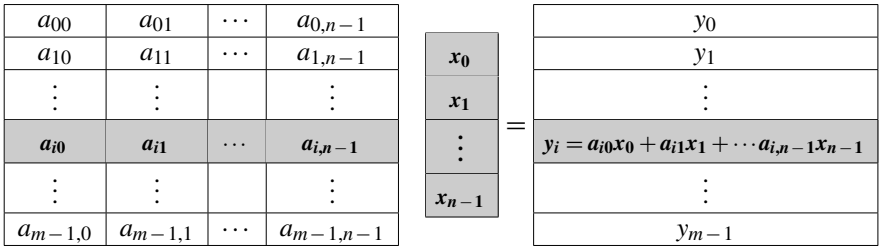


FIGURE 4.3

Matrix-vector multiplication

the dot product of the ith row of A with $\mathbf{x}$:

$$y_i = \sum_{j=0}^{n-1} a_{ij}x_j.$$

See Figure 4.3. Thus, pseudo-code for a *serial* program for matrix-vector multiplication might look like this:

```
/* For each row of A */
for (i = 0; i < m; i++) {
   y[i] = 0.0;
   /* For each element of the row and each element of x */
   for (j = 0; j < n; j++)
      y[i] += A[i][j]* x[j];
}
```

We want to parallelize this by dividing the work among the threads. One possibility is to divide the iterations of the outer loop among the threads. If we do this, each thread will compute some of the components of `y`. For example, suppose that $m = n = 6$ and the number of threads, `thread_count` or t, is three. Then the computation could be divided among the threads as follows:

Thread	Components of y
0	y[0], y[1]
1	y[2], y[3]
2	y[4], y[5]

To compute `y[0]`, thread 0 will need to execute the code

```
y[0] = 0.0;
for (j = 0; j < n; j++)
   y[0] += A[0][j]* x[j];
```

Thread 0 will therefore need to access every element of row 0 of `A` and every element of `x`. More generally, the thread that has been assigned `y[i]` will need to execute the code

```
y[i] = 0.0;
for (j = 0; j < n; j++)
   y[i] += A[i][j]*x[j];
```

Thus, this thread will need to access every element of row `i` of `A` and every element of `x`. We see that each thread needs to access every component of `x`, while each thread only needs to access its assigned rows of `A` and assigned components of `y`. This suggests that, at a minimum, `x` should be shared. Let's also make `A` and `y` shared. This might seem to violate our principle that we should only make variables global that need to be global. However, in the exercises, we'll take a closer look at some of the issues involved in making the `A` and `y` variables local to the thread function, and we'll see that making them global can make good sense. At this point, we'll just observe that if they are global, the main thread can easily initialize all of `A` by just reading its entries from `stdin`, and the product vector `y` can be easily printed by the main thread.

Having made these decisions, we only need to write the code that each thread will use for deciding which components of `y` it will compute. In order to simplify the code, let's assume that both m and n are evenly divisible by t. Our example with $m = 6$ and $t = 3$ suggests that each thread gets m/t components. Furthermore, thread 0 gets the first m/t, thread 1 gets the next m/t, and so on. Thus, the formulas for the components assigned to thread q might be

$$\text{first component: } q \times \frac{m}{t}$$

and

$$\text{last component: } (q+1) \times \frac{m}{t} - 1.$$

With these formulas, we can write the thread function that carries out matrix-vector multiplication. See Program 4.2. Note that in this code, we're assuming that `A`, `x`, `y`, `m`, and `n` are all global and shared.

```
void* Pth_mat_vect(void* rank) {
   long my_rank = (long) rank;
   int i, j;
   int local_m = m/thread_count;
   int my_first_row = my_rank*local_m;
   int my_last_row = (my_rank+1)*local_m - 1;

   for (i = my_first_row; i <= my_last_row; i++) {
      y[i] = 0.0;
      for (j = 0; j < n; j++)
          y[i] += A[i][j]*x[j];
   }

   return NULL;
}  /* Pth_mat_vect */
```

Program 4.2: Pthreads matrix-vector multiplication

If you have already read the MPI chapter, you may recall that it took more work to write a matrix-vector multiplication program using MPI. This was because of the fact that the data structures were necessarily distributed, that is, each MPI process only has direct access to its own local memory. Thus, for the MPI code, we need to explicitly *gather* all of `x` into each process' memory. We see from this example that there are instances in which writing shared-memory programs is easier than writing distributed-memory programs. However, we'll shortly see that there are situations in which shared-memory programs can be more complex.

4.4 CRITICAL SECTIONS

Matrix-vector multiplication was very easy to code because the shared-memory locations were accessed in a highly desirable way. After initialization, all of the variables—except `y`—are only *read* by the threads. That is, except for `y`, none of the shared variables are changed after they've been initialized by the main thread. Furthermore, although the threads do make changes to `y`, only one thread makes changes to any individual component, so there are no attempts by two (or more) threads to modify any single component. What happens if this isn't the case? That is, what happens when multiple threads update a single memory location? We also discuss this in Chapters 2 and 5, so if you've read one of these chapters, you already know the answer. But let's look at an example.

Let's try to estimate the value of π. There are lots of different formulas we could use. One of the simplest is

$$\pi = 4\left(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots + (-1)^n \frac{1}{2n+1} + \cdots\right).$$

This isn't the best formula for computing π, because it takes *a lot* of terms on the right-hand side before it is very accurate. However, for our purposes, lots of terms will be better.

The following *serial* code uses this formula:

```
double factor = 1.0;
double sum = 0.0;
for (i = 0; i < n; i++, factor = -factor) {
   sum += factor/(2*i+1);
}
pi = 4.0*sum;
```

We can try to parallelize this in the same way we parallelized the matrix-vector multiplication program: divide up the iterations in the `for` loop among the threads and make `sum` a shared variable. To simplify the computations, let's assume that the number of threads, `thread_count` or t, evenly divides the number of terms in the sum, n. Then, if $\bar{n} = n/t$, thread 0 can add the first $\bar{n}$ terms. Therefore, for thread 0, the loop variable `i` will range from 0 to $\bar{n} - 1$. Thread 1 will add the next $\bar{n}$ terms, so for thread 1, the loop variable will range from $\bar{n}$ to $2\bar{n} - 1$. More generally, for thread q the loop

```
1  void* Thread_sum(void* rank) {
2     long my_rank = (long) rank;
3     double factor;
4     long long i;
5     long long my_n = n/thread_count;
6     long long my_first_i = my_n*my_rank;
7     long long my_last_i = my_first_i + my_n;
8
9     if (my_first_i % 2 == 0)  /* my_first_i is even */
10       factor = 1.0;
11    else  /* my_first_i is odd */
12       factor = -1.0;
13
14    for (i = my_first_i; i < my_last_i; i++, factor = -factor) {
15       sum += factor/(2*i+1);
16    }
17
18    return NULL;
19 }  /* Thread_sum */
```

Program 4.3: An attempt at a thread function for computing π

variable will range over

$$q\bar{n}, q\bar{n}+1, q\bar{n}+2, \ldots, (q+1)\bar{n}-1.$$

Furthermore, the sign of the first term, term $q\bar{n}$, will be positive if $q\bar{n}$ is even and negative if $q\bar{n}$ is odd. The thread function might use the code shown in Program 4.3.

If we run the Pthreads program with two threads and n is relatively small, we find that the results of the Pthreads program are in agreement with the serial sum program. However, as n gets larger, we start getting some peculiar results. For example, with a dual-core processor we get the following results:

	n			
	10^5	10^6	10^7	10^8
π	3.14159	3.141593	3.1415927	3.14159265
1 Thread	3.14158	3.141592	3.1415926	3.14159264
2 Threads	3.14158	3.141480	3.1413692	3.14164686

Notice that as we increase n, the estimate with one thread gets better and better. In fact, with each factor of 10 increase in n we get another correct digit. With $n = 10^5$, the result as computed by a single thread has five correct digits. With $n = 10^6$, it has six correct digits, and so on. The result computed by two threads agrees with the

result computed by one thread when $n = 10^5$. However, for larger values of n, the result computed by two threads actually gets worse. In fact, if we ran the program several times with two threads and the same value of n, we would see that the result computed by two threads *changes* from run to run. The answer to our original question must clearly be, "Yes, it matters if multiple threads try to update a single shared variable."

Let's recall why this is the case. Remember that the addition of two values is typically *not* a single machine instruction. For example, although we can add the contents of a memory location `y` to a memory location `x` with a single C statement,

```
x = x + y;
```

what the machine does is typically more complicated. The current values stored in `x` and `y` will, in general, be stored in the computer's main memory, which has no circuitry for carrying out arithmetic operations. Before the addition can be carried out, the values stored in `x` and `y` may therefore have to be transferred from main memory to registers in the CPU. Once the values are in registers, the addition can be carried out. After the addition is completed, the result may have to be transferred from a register back to memory.

Suppose that we have two threads, and each computes a value that is stored in its private variable `y`. Also suppose that we want to add these private values together into a shared variable `x` that has been initialized to 0 by the main thread. Each thread will execute the following code:

```
y = Compute(my_rank);
x = x + y;
```

Let's also suppose that thread 0 computes `y = 1` and thread 1 computes `y = 2`. The "correct" result should then be `x = 3`. Here's one possible scenario:

Time	Thread 0	Thread 1
1	Started by main thread	
2	Call `Compute()`	Started by main thread
3	Assign `y = 1`	Call `Compute()`
4	Put `x=0` and `y=1` into registers	Assign `y = 2`
5	Add 0 and 1	Put `x=0` and `y=2` into registers
6	Store 1 in memory location `x`	Add 0 and 2
7		Store 2 in memory location `x`

We see that if thread 1 copies `x` from memory to a register *before* thread 0 stores its result, the computation carried out by thread 0 will be *overwritten* by thread 1. The problem could be reversed: if thread 1 *races* ahead of thread 0, then its result may be overwritten by thread 0. In fact, unless one of the threads stores its result *before* the other thread starts reading `x` from memory, the "winner's" result will be overwritten by the "loser."

This example illustrates a fundamental problem in shared-memory programming: when multiple threads attempt to update a shared resource—in our case a shared variable—the result may be unpredictable. Recall that more generally, when multiple threads attempt to access a shared resource such as a shared variable or a shared file, at least one of the accesses is an update, and the accesses can result in an error, we have a **race condition**. In our example, in order for our code to produce the correct result, we need to make sure that once one of the threads starts executing the statement `x = x + y`, it finishes executing the statement *before* the other thread starts executing the statement. Therefore, the code `x = x + y` is a **critical section**, that is, it's a block of code that updates a shared resource that can only be updated by one thread at a time.

4.5 BUSY-WAITING

When, say, thread 0 wants to execute the statement `x = x + y`, it needs to first make sure that thread 1 is not already executing the statement. Once thread 0 makes sure of this, it needs to provide some way for thread 1 to determine that it, thread 0, is executing the statement, so that thread 1 won't attempt to start executing the statement until thread 0 is done. Finally, after thread 0 has completed execution of the statement, it needs to provide some way for thread 1 to determine that it is done, so that thread 1 can safely start executing the statement.

A simple approach that doesn't involve any new concepts is the use of a flag variable. Suppose `flag` is a shared `int` that is set to 0 by the main thread. Further, suppose we add the following code to our example:

```
1  y = Compute(my_rank);
2  while (flag != my_rank);
3  x = x + y;
4  flag++;
```

Let's suppose that thread 1 finishes the assignment in Line 1 before thread 0. What happens when it reaches the `while` statement in Line 2? If you look at the `while` statement for a minute, you'll see that it has the somewhat peculiar property that its body is empty. So if the test `flag != my_rank` is true, then thread 1 will just execute the test a second time. In fact, it will keep re-executing the test until the test is false. When the test is false, thread 1 will go on to execute the code in the critical section `x = x + y`.

Since we're assuming that the main thread has initialized `flag` to 0, thread 1 won't proceed to the critical section in Line 3 until thread 0 executes the statement `flag++`. In fact, we see that unless some catastrophe befalls thread 0, it will eventually catch up to thread 1. However, when thread 0 executes its first test of `flag != my_rank`, the condition is false, and it will go on to execute the code in the critical section `x = x + y`. When it's done with this, we see that it will execute `flag++`, and thread 1 can finally enter the critical section.

The key here is that thread 1 *cannot enter the critical section until thread 0 has completed the execution of* `flag++`. And, provided the statements are executed exactly as they're written, this means that thread 1 cannot enter the critical section until thread 0 has completed it.

The `while` loop is an example of **busy-waiting**. In busy-waiting, a thread repeatedly tests a condition, but, effectively, does no useful work until the condition has the appropriate value (false in our example).

Note that we said that the busy-wait solution would work "provided the statements are executed exactly as they're written." If compiler optimization is turned on, it *is* possible that the compiler will make changes that will affect the correctness of busy-waiting. The reason for this is that the compiler is unaware that the program is multithreaded, so it doesn't "know" that the variables `x` and `flag` can be modified by another thread. For example, if our code

```
y = Compute(my_rank);
while (flag != my_rank);
x = x + y;
flag++;
```

is run by just one thread, the order of the statements `while (flag != my_rank)` and `x = x + y` is unimportant. An optimizing compiler might therefore determine that the program would make better use of registers if the order of the statements were switched. Of course, this will result in the code

```
y = Compute(my_rank);
x = x + y;
while (flag != my_rank);
flag++;
```

which defeats the purpose of the busy-wait loop. The simplest solution to this problem is to turn compiler optimizations off when we use busy-waiting. For an alternative to completely turning off optimizations, see Exercise 4.3.

We can immediately see that busy-waiting is not an ideal solution to the problem of controlling access to a critical section. Since thread 1 will execute the test over and over until thread 0 executes `flag++`, if thread 0 is delayed (for example, if the operating system preempts it to run something else), thread 1 will simply "spin" on the test, eating up CPU cycles. This can be positively disastrous for performance. Turning off compiler optimizations can also seriously degrade performance.

Before going on, though, let's return to our π calculation program in Figure 4.3 and correct it by using busy-waiting. The critical section in this function is Line 15. We can therefore precede this with a busy-wait loop. However, when a thread is done with the critical section, if it simply increments `flag`, eventually `flag` will be greater than t, the number of threads, and none of the threads will be able to return to the critical section. That is, after executing the critical section once, all the threads will be stuck forever in the busy-wait loop. Thus, in this instance, we don't want to simply

```
1  void* Thread_sum(void* rank) {
2     long my_rank = (long) rank;
3     double factor;
4     long long i;
5     long long my_n = n/thread_count;
6     long long my_first_i = my_n*my_rank;
7     long long my_last_i = my_first_i + my_n;
8
9     if (my_first_i % 2 == 0)
10       factor = 1.0;
11    else
12       factor = -1.0;
13
14    for (i = my_first_i; i < my_last_i; i++, factor = -factor) {
15       while (flag != my_rank);
16       sum += factor/(2*i+1);
17       flag = (flag+1) % thread_count;
18    }
19
20    return NULL;
21 }  /* Thread_sum */
```

Program 4.4: Pthreads global sum with busy-waiting

increment `flag`. Rather, the last thread, thread $t-1$, should reset `flag` to zero. This can be accomplished by replacing `flag++` with

```
flag = (flag + 1) % thread_count;
```

With this change, we get the thread function shown in Program 4.4. If we compile the program and run it with two threads, we see that it is computing the correct results. However, if we add in code for computing elapsed time, we see that when $n = 10^8$, the serial sum is consistently faster than the parallel sum. For example, on the dual-core system, the elapsed time for the sum as computed by two threads is about 19.5 seconds, while the elapsed time for the serial sum is about 2.8 seconds!

Why is this? Of course, there's overhead associated with starting up and joining the threads. However, we can estimate this overhead by writing a Pthreads program in which the thread function simply returns:

```
void* Thread_function(void* ignore) {
   return NULL;
}  /* Thread_function */
```

When we find the time that's elapsed between starting the first thread and joining the second thread, we see that on this particular system, the overhead is less than 0.3 milliseconds, so the slowdown isn't due to thread overhead. If we look closely at

```
void* Thread_sum(void* rank) {
   long my_rank = (long) rank;
   double factor, my_sum = 0.0;
   long long i;
   long long my_n = n/thread_count;
   long long my_first_i = my_n*my_rank;
   long long my_last_i = my_first_i + my_n;

   if (my_first_i % 2 == 0)
      factor = 1.0;
   else
      factor = -1.0;

   for (i = my_first_i; i < my_last_i; i++, factor = -factor)
      my_sum += factor/(2*i+1);

   while (flag != my_rank);
   sum += my_sum;
   flag = (flag+1) % thread_count;

   return NULL;
}  /* Thread_sum */
```

Program 4.5: Global sum function with critical section after loop

the thread function that uses busy-waiting, we see that the threads alternate between executing the critical section code in Line 16. Initially `flag` is 0, so thread 1 must wait until thread 0 executes the critical section and increments `flag`. Then, thread 0 must wait until thread 1 executes and increments. The threads will alternate between waiting and executing, and evidently the waiting and the incrementing increase the overall run time by a factor of seven.

As we'll see, busy-waiting isn't the only solution to protecting a critical section. In fact, there are much better solutions. However, since the code in a critical section can only be executed by one thread at a time, no matter how we limit access to the critical section, we'll effectively serialize the code in the critical section. Therefore, if it's at all possible, we should minimize the number of times we execute critical section code. One way to greatly improve the performance of the sum function is to have each thread use a *private* variable to store its total contribution to the sum. Then, each thread can add in its contribution to the global sum once, *after* the `for` loop. See Program 4.5. When we run this on the dual-core system with $n = 10^8$, the elapsed time is reduced to 1.5 seconds for two threads, a *substantial* improvement.

4.6 MUTEXES

Since a thread that is busy-waiting may continually use the CPU, busy-waiting is generally not an ideal solution to the problem of limiting access to a critical section. Two

better solutions are mutexes and semaphores. **Mutex** is an abbreviation of *mutual exclusion,* and a mutex is a special type of variable that, together with a couple of special functions, can be used to restrict access to a critical section to a single thread at a time. Thus, a mutex can be used to guarantee that one thread "excludes" all other threads while it executes the critical section. Hence, the mutex guarantees mutually exclusive access to the critical section.

The Pthreads standard includes a special type for mutexes: `pthread_mutex_t`. A variable of type `pthread_mutex_t` needs to be initialized by the system before it's used. This can be done with a call to

```
int pthread_mutex_init(
      pthread_mutex_t*            mutex_p   /* out */,
      const pthread_mutexattr_t*  attr_p    /* in  */);
```

We won't make use of the second argument, so we'll just pass in `NULL`. When a Pthreads program finishes using a mutex, it should call

```
int pthread_mutex_destroy(pthread_mutex_t* mutex_p  /* in/out */);
```

To gain access to a critical section, a thread calls

```
int pthread_mutex_lock(pthread_mutex_t* mutex_p  /* in/out */);
```

When a thread is finished executing the code in a critical section, it should call

```
int pthread_mutex_unlock(pthread_mutex_t* mutex_p  /* in/out */);
```

The call to `pthread_mutex_lock` will cause the thread to wait until no other thread is in the critical section, and the call to `pthread_mutex_unlock` notifies the system that the calling thread has completed execution of the code in the critical section.

We can use mutexes instead of busy-waiting in our global sum program by declaring a global mutex variable, having the main thread initialize it, and then, instead of busy-waiting and incrementing a flag, the threads call `pthread_mutex_lock` before entering the critical section, and they call `pthread_mutex_unlock` when they're done with the critical section. See Program 4.6. The first thread to call `pthread_mutex_lock` will, effectively, "lock the door" to the critical section. Any other thread that attempts to execute the critical section code must first also call `pthread_mutex_lock`, and until the first thread calls `pthread_mutex_unlock`, all the threads that have called `pthread_mutex_lock` will **block** in their calls—they'll just wait until the first thread is done. After the first thread calls `pthread_mutex_unlock`, the system will choose one of the blocked threads and allow it to execute the code in the critical section. This process will be repeated until all the threads have completed executing the critical section.

"Locking" and "unlocking" the door to the critical section isn't the only metaphor that's used in connection with mutexes. Programmers often say that the thread that has returned from a call to `pthread_mutex_lock` has "obtained the mutex" or "obtained the lock." When this terminology is used, a thread that calls `pthread_mutex_unlock` "relinquishes" the mutex or lock.

```
1  void* Thread_sum(void* rank) {
2     long my_rank = (long) rank;
3     double factor;
4     long long i;
5     long long my_n = n/thread_count;
6     long long my_first_i = my_n*my_rank;
7     long long my_last_i = my_first_i + my_n;
8     double my_sum = 0.0;
9
10    if (my_first_i % 2 == 0)
11       factor = 1.0;
12    else
13       factor = -1.0;
14
15    for (i = my_first_i; i < my_last_i; i++, factor = -factor) {
16       my_sum += factor/(2*i+1);
17    }
18    pthread_mutex_lock(&mutex);
19    sum += my_sum;
20    pthread_mutex_unlock(&mutex);
21
22    return NULL;
23 }  /* Thread_sum */
```

Program 4.6: Global sum function that uses a mutex

Notice that with mutexes (unlike our busy-waiting solution), the order in which the threads execute the code in the critical section is more or less random: the first thread to call `pthread_mutex_lock` will be the first to execute the code in the critical section. Subsequent accesses will be scheduled by the system. Pthreads doesn't guarantee (for example) that the threads will obtain the lock in the order in which they called `Pthread_mutex_lock`. However, in our setting, only finitely many threads will try to acquire the lock. Eventually each thread will obtain the lock.

If we look at the (unoptimized) performance of the busy-wait π program (with the critical section after the loop) and the mutex program, we see that for both versions the ratio of the run-time of the single-threaded program with the multithreaded program is equal to the number of threads, as long as the number of threads is no greater than the number of cores. (See Table 4.1.) That is,

$$\frac{T_{\text{serial}}}{T_{\text{parallel}}} \approx \texttt{thread\_count},$$

provided `thread_count` is less than or equal to the number of cores. Recall that $T_{\text{serial}}/T_{\text{parallel}}$ is called the *speedup*, and when the speedup is equal to the number of threads, we have achieved more or less "ideal" performance or *linear speedup*.

If we compare the performance of the version that uses busy-waiting with the version that uses mutexes, we don't see much difference in the overall run-time when the programs are run with fewer threads than cores. This shouldn't be surprising,

Table 4.1 Run-Times (in Seconds) of π Programs Using $n = 10^8$ Terms on a System with Two Four-Core Processors

Threads	Busy-Wait	Mutex
1	2.90	2.90
2	1.45	1.45
4	0.73	0.73
8	0.38	0.38
16	0.50	0.38
32	0.80	0.40
64	3.56	0.38

as each thread only enters the critical section once; so unless the critical section is very long, or the Pthreads functions are very slow, we wouldn't expect the threads to be delayed very much by waiting to enter the critical section. However, if we start increasing the number of threads beyond the number of cores, the performance of the version that uses mutexes remains pretty much unchanged, while the performance of the busy-wait version degrades.

We see that when we use busy-waiting, performance can degrade if there are more threads than cores.[4] This should make sense. For example, suppose we have two cores and five threads. Also suppose that thread 0 is in the critical section, thread 1 is in the busy-wait loop, and threads 2, 3, and 4 have been descheduled by the operating system. After thread 0 completes the critical section and sets `flag = 1`, it will be terminated, and thread 1 can enter the critical section so the operating system can schedule thread 2, thread 3, or thread 4. Suppose it schedules thread 3, which will spin in the `while` loop. When thread 1 finishes the critical section and sets `flag = 2`, the operating system can schedule thread 2 or thread 4. If it schedules thread 4, then both thread 3 and thread 4, will be busily spinning in the busy-wait loop until the operating system deschedules one of them and schedules thread 2. See Table 4.2.

4.7 PRODUCER-CONSUMER SYNCHRONIZATION AND SEMAPHORES

Although busy-waiting is generally wasteful of CPU resources, it has the property by which we know, in advance, the order in which the threads will execute the code in the critical section: thread 0 is first, then thread 1, then thread 2, and so on. With

[4]These are typical run-times. When using busy-waiting and the number of threads is greater than the number of cores, the run-times vary considerably.

Table 4.2 Possible Sequence of Events with Busy-Waiting and More Threads than Cores

		Thread				
Time	`flag`	**0**	**1**	**2**	**3**	**4**
0	0	crit sect	busy-wait	susp	susp	susp
1	1	terminate	crit sect	susp	busy-wait	susp
2	2	—	terminate	susp	busy-wait	busy-wait
⋮	⋮			⋮	⋮	⋮
?	2	—	—	crit sect	susp	busy-wait

mutexes, the order in which the threads execute the critical section is left to chance and the system. Since addition is commutative, this doesn't matter in our program for estimating π. However, it's not difficult to think of situations in which we also want to control the order in which the threads execute the code in the critical section. For example, suppose each thread generates an $n \times n$ matrix, and we want to multiply the matrices together in thread-rank order. Since matrix multiplication isn't commutative, our mutex solution would have problems:

```
/* n and product_matrix are shared and initialized by the main
thread */
/* product_matrix is initialized to be the identity matrix  */
void* Thread_work(void* rank) {
   long my_rank = (long) rank;
   matrix_t my_mat = Allocate_matrix(n);
   Generate_matrix(my_mat);
   pthread_mutex_lock(&mutex);
   Multiply_matrix(product_mat, my_mat);
   pthread_mutex_unlock(&mutex);
   Free_matrix(&my_mat);
   return NULL;
}  /* Thread_work */
```

A somewhat more complicated example involves having each thread "send a message" to another thread. For example, suppose we have `thread_count` or t threads and we want thread 0 to send a message to thread 1, thread 1 to send a message to thread 2, ..., thread $t-2$ to send a message to thread $t-1$ and thread $t-1$ to send a message to thread 0. After a thread "receives" a message, it can print the message and terminate. In order to implement the message transfer, we can allocate a shared array of `char*`. Then each thread can allocate storage for the message it's sending, and, after it has initialized the message, set a pointer in the shared array to refer to it. In order to avoid dereferencing undefined pointers, the main thread can set the individual entries in the shared array to `NULL`. See Program 4.7. When we run the program with more than a couple of threads on a dual-core system, we see that some of the messages are never received. For example, thread 0, which is started first,

```
1  /* messages has type char**. It's allocated in main. */
2  /* Each entry is set to NULL in main.                */
3  void* Send_msg(void* rank) {
4     long my_rank = (long) rank;
5     long dest = (my_rank + 1) % thread_count;
6     long source = (my_rank + thread_count - 1) % thread_count;
7     char* my_msg = malloc(MSG_MAX*sizeof(char));
8
9     sprintf(my_msg, "Hello to %ld from %ld", dest, my_rank);
10    messages[dest] = my_msg;
11
12    if (messages[my_rank] != NULL)
13       printf("Thread %ld > %s\n", my_rank, messages[my_rank]);
14    else
15       printf("Thread %ld > No message from %ld\n", my_rank,
               source);
16
17    return NULL;
18 }  /* Send_msg */
```

Program 4.7: A first attempt at sending messages using Pthreads

will typically finish before thread $t-1$ has copied the message into the `messages` array. This isn't surprising, and we could fix the problem by replacing the `if` statement in Line 12 with a busy-wait `while` statement:

```
while (messages[my_rank] == NULL);
printf("Thread %ld > %s\n", my_rank, messages[my_rank]);
```

Of course, this solution would have the same problems that any busy-waiting solution has, so we'd prefer a different approach.

After executing the assignment in Line 10, we'd like to "notify" the thread with rank `dest` that it can proceed to print the message. We'd like to do something like this:

```
. . .
messages[dest] = my_msg;
Notify thread dest that it can proceed;

Await notification from thread source
printf("Thread %ld > %s\n", my_rank, messages[my_rank]);
. . .
```

It's not at all clear how mutexes can be of help here. We might try calling `pthread_mutex_unlock` to "notify" the thread with rank `dest`. However, mutexes are initialized to be *unlocked*, so we'd need to add a call *before* initializing `messages[dest]` to lock the mutex. This will be a problem since we don't know when the threads will reach the calls to `pthread_mutex_lock`.

To make this a little clearer, suppose that the main thread creates and initializes an array of mutexes, one for each thread. Then, we're trying to do something like this:

```
1  . . .
2  pthread_mutex_lock(mutex[dest]);
3  . . .
4  messages[dest] = my_msg;
5  pthread_mutex_unlock(mutex[dest]);
6  . . .
7  pthread_mutex_lock(mutex[my_rank]);
8  printf("Thread %ld > %s\n", my_rank, messages[my_rank]);
9  . . .
```

Now suppose we have two threads, and thread 0 gets so far ahead of thread 1 that it reaches the second call to `pthread_mutex_lock` in Line 7 before thread 1 reaches the first in Line 2. Then, of course, it will acquire the lock and continue to the `printf` statement. This will result in thread 0's dereferencing a null pointer, and it will crash.

There *are* other approaches to solving this problem with mutexes. See, for example, Exercise 4.7. However, POSIX also provides a somewhat different means of controlling access to critical sections: **semaphores**. Let's take a look at them.

Semaphores can be thought of as a special type of `unsigned int`, so they can take on the values 0, 1, 2, ... In most cases, we'll only be interested in using them when they take on the values 0 and 1. A semaphore that only takes on these values is called a *binary* semaphore. Very roughly speaking, 0 corresponds to a locked mutex, and 1 corresponds to an unlocked mutex. To use a binary semaphore as a mutex, you *initialize* it to 1—that is, it's "unlocked." Before the critical section you want to protect, you place a call to the function `sem_wait`. A thread that executes `sem_wait` will block if the semaphore is 0. If the semaphore is nonzero, it will *decrement* the semaphore and proceed. After executing the code in the critical section, a thread calls `sem_post`, which *increments* the semaphore, and a thread waiting in `sem_wait` can proceed.

Semaphores were first defined by the computer scientist Edsger Dijkstra in [13]. The name is taken from the mechanical device that railroads use to control which train can use a track. The device consists of an arm attached by a pivot to a post. When the arm points down, approaching trains can proceed, and when the arm is perpendicular to the post, approaching trains must stop and wait. The track corresponds to the critical section: when the arm is down corresponds to a semaphore of 1, and when the arm is up corresponds to a semaphore of 0. The `sem_wait` and `sem_post` calls correspond to signals sent by the train to the semaphore controller.

For our current purposes, the crucial difference between semaphores and mutexes is that there is no ownership associated with a semaphore. The main thread can initialize all of the semaphores to 0—that is, "locked," and then any thread can execute a `sem_post` on any of the semaphores, and, similarly, any thread can execute `sem_wait` on any of the semaphores. Thus, if we use semaphores, our `Send_msg` function can be written as shown in Program 4.8.

```
1  /* messages is allocated and initialized to NULL in main   */
2  /* semaphores is allocated and initialized to 0 (locked) in
        main */
3  void* Send_msg(void* rank) {
4     long my_rank = (long) rank;
5     long dest = (my_rank + 1) % thread_count;
6     char* my_msg = malloc(MSG_MAX*sizeof(char));
7
8     sprintf(my_msg, "Hello to %ld from %ld", dest, my_rank);
9     messages[dest] = my_msg;
10    sem_post(&semaphores[dest])
            /* ''Unlock'' the semaphore of dest */
11
12    /* Wait for our semaphore to be unlocked */
13    sem_wait(&semaphores[my_rank]);
14    printf("Thread %ld > %s\n", my_rank, messages[my_rank]);
15
16    return NULL;
17 }  /* Send_msg */
```

Program 4.8: Using semaphores so that threads can send messages

The syntax of the various semaphore functions is

```
int sem_init(
      sem_t*    semaphore_p   /* out */,
      int       shared        /* in  */,
      unsigned  initial_val   /* in  */);

int sem_destroy(sem_t*  semaphore_p  /* in/out */);
int sem_post(sem_t*     semaphore_p  /* in/out */);
int sem_wait(sem_t*     semaphore_p  /* in/out */);
```

We won't make use of the second argument to `sem_init`: the constant 0 can be passed in. Note that semaphores are *not* part of Pthreads. Hence, it's necessary to add the following preprocessor directive to any program that uses them:[5]

```
#include <semaphore.h>
```

Finally, note that the message-sending problem didn't involve a critical section. The problem wasn't that there was a block of code that could only be executed by one thread at a time. Rather, thread `my_rank` couldn't proceed until thread `source` had finished creating the message. This type of synchronization, when a thread can't

[5]Some systems (e.g., some versions of Mac OS X) don't support this version of semaphores. They support something called "named" semaphores. The functions `sem_wait` and `sem_post` can be used in the same way. However, `sem_init` should be replaced by `sem_open`, and `sem_destroy` should be replaced by `sem_close` and `sem_unlink`. See the book's website for an example.

proceed until another thread has taken some action, is sometimes called **producer-consumer synchronization**.

4.8 BARRIERS AND CONDITION VARIABLES

Let's take a look at another problem in shared-memory programming: synchronizing the threads by making sure that they all are at the same point in a program. Such a point of synchronization is called a **barrier** because no thread can proceed beyond the barrier until all the threads have reached it.

Barriers have numerous applications. As we discussed in Chapter 2, if we're timing some part of a multithreaded program, we'd like for all the threads to start the timed code at the same instant, and then report the time taken by the last thread to finish, that is, the "slowest" thread. We'd therefore like to do something like this:

```
/* Shared */
double elapsed_time;
. . .
/* Private */
double my_start, my_finish, my_elapsed;
. . .
Synchronize threads;
Store current time in my_start;
/* Execute timed code */
. . .
Store current time in my_finish;
my_elapsed = my_finish - my_start;

elapsed = Maximum of my_elapsed values;
```

Using this approach, we're sure that all of the threads will record `my_start` at approximately the same time.

Another very important use of barriers is in debugging. As you've probably already seen, it can be very difficult to determine *where* an error is occuring in a parallel program. We can, of course, have each thread print a message indicating which point it's reached in the program, but it doesn't take long for the volume of the output to become overwhelming. Barriers provide an alternative:

```
point in program we want to reach;
barrier;
if (my_rank == 0) {
   printf("All threads reached this point\n");
   fflush(stdout);
}
```

Many implementations of Pthreads don't provide barriers, so if our code is to be portable, we need to develop our own implementation. There are a number of options; we'll look at three. The first two only use constructs that we've already studied. The third uses a new type of Pthreads object: a *condition variable.*

4.8.1 Busy-waiting and a mutex

Implementing a barrier using busy-waiting and a mutex is straightforward: we use a shared counter protected by the mutex. When the counter indicates that every thread has entered the critical section, threads can leave a busy-wait loop.

```
/* Shared and initialized by the main thread */
int counter;  /* Initialize to 0 */
int thread_count;
pthread_mutex_t barrier_mutex;
. . .

void* Thread_work(. . .) {
   . . .
   /* Barrier */
   pthread_mutex_lock(&barrier_mutex);
   counter++;
   pthread_mutex_unlock(&barrier_mutex);
   while (counter < thread_count);
   . . .
}
```

Of course, this implementation will have the same problems that our other busy-wait codes had: we'll waste CPU cycles when threads are in the busy-wait loop, and, if we run the program with more threads than cores, we may find that the performance of the program seriously degrades.

Another issue is the shared variable `counter`. What happens if we want to implement a second barrier and we try to reuse the counter? When the first barrier is completed, `counter` will have the value `thread_count`. Unless we can somehow reset `counter`, the `while` condition we used for our first barrier `counter < thread_count` will be false, and the barrier won't cause the threads to block. Furthermore, any attempt to reset `counter` to zero is almost certainly doomed to failure. If the last thread to enter the loop tries to reset it, some thread in the busy-wait may never see the fact that `counter == thread_count`, and that thread may hang in the busy-wait. If some thread tries to reset the counter after the barrier, some other thread may enter the second barrier before the counter is reset and its increment to the counter will be lost. This will have the unfortunate effect of causing all the threads to hang in the second busy-wait loop. So if we want to use this barrier, we need one counter variable for each instance of the barrier.

4.8.2 Semaphores

A natural question is whether we can implement a barrier with semaphores, and, if so, whether we can reduce the number of problems we encountered with busy-waiting. The answer to the first question is yes:

```
/* Shared variables */
int counter;        /* Initialize to 0 */
```

```
sem_t count_sem;    /* Initialize to 1 */
sem_t barrier_sem;  /* Initialize to 0 */
. . .
void* Thread_work(...) {
   . . .
   /* Barrier */
   sem_wait(&count_sem);
   if (counter == thread_count-1) {
      counter = 0;
      sem_post(&count_sem);
      for (j = 0; j < thread_count-1; j++)
         sem_post(&barrier_sem);
   } else {
      counter++;
      sem_post(&count_sem);
      sem_wait(&barrier_sem);
   }
   . . .
}
```

As with the busy-wait barrier, we have a counter that we use to determine how many threads have entered the barrier. We use two semaphores: `count_sem` protects the counter, and `barrier_sem` is used to block threads that have entered the barrier. The `count_sem` semaphore is initialized to 1 (that is, "unlocked"), so the first thread to reach the barrier will be able to proceed past the call to `sem_wait`. Subsequent threads, however, will block until they can have exclusive access to the counter. When a thread has exclusive access to the counter, it checks to see if `counter < thread_count`-1. If it is, the thread increments `counter` "relinquishes the lock" (`sem_post(&count_sem)`) and blocks in `sem_wait(&barrier_sem)`. On the other hand, if `counter == thread_count`-1, the thread is the last to enter the barrier, so it can reset `counter` to zero and "unlock" `count_sem` by calling `sem_post(&count_sem)`. Now, it wants to notify all the other threads that they can proceed, so it executes `sem_post(&barrier_sem)` for each of the `thread_count`-1 threads that are blocked in `sem_wait(&barrier_sem)`.

Note that it doesn't matter if the thread executing the loop of calls to `sem_post(&barrier_sem)` races ahead and executes multiple calls to `sem_post` before a thread can be unblocked from `sem_wait(&barrier_sem)`. For recall that a semaphore is an **unsigned** `int`, and the calls to `sem_post` increment it, while the calls to `sem_wait` decrement it—unless it's already 0, in which case the calling threads will block until it's positive again, and they'll decrement it when they unblock. Therefore, it doesn't matter if the thread executing the loop of calls to `sem_post(&barrier_sem)` gets ahead of the threads blocked in the calls to `sem_wait(&barrier_sem)`, because eventually the blocked threads will see that `barrier_sem` is positive, and they'll decrement it and proceed.

It should be clear that this implementation of a barrier is superior to the busy-wait barrier, since the threads don't need to consume CPU cycles when they're blocked

in sem_wait. Can we reuse the data structures from the first barrier if we want to execute a second barrier?

The counter can be reused, since we were careful to reset it before releasing any of the threads from the barrier. Also, count_sem can be reused, since it is reset to 1 before any threads can leave the barrier. This leaves barrier_sem. Since there's exactly one sem_post for each sem_wait, it might appear that the value of barrier_sem will be 0 when the threads start executing a second barrier. However, suppose we have two threads, and thread 0 is blocked in sem_wait(&barrier_sem) in the first barrier, while thread 1 is executing the loop of sem_post. Also suppose that the operating system has seen that thread 0 is idle, and descheduled it out. Then thread 1 can go on to the second barrier. Since counter == 0, it will execute the else clause. After incrementing counter, it executes sem_post(&count_sem), and then executes sem_wait(&barrier_sem).

However, if thread 0 is still descheduled, it will not have decremented barrier_sem. Thus when thread 1 reaches sem_wait(&barrier_sem), barrier_sem will still be 1, so it will simply decrement barrier_sem and proceed. This will have the unfortunate consequence that when thread 0 starts executing again, it will still be blocked in the *first* sem_wait(&barrier_sem), and thread 1 will proceed through the second barrier before thread 0 has entered it. Reusing barrier_sem therefore results in a race condition.

4.8.3 Condition variables

A somewhat better approach to creating a barrier in Pthreads is provided by *condition variables.* A **condition variable** is a data object that allows a thread to suspend execution until a certain event or *condition* occurs. When the event or condition occurs another thread can *signal* the thread to "wake up." A condition variable is *always* associated with a mutex.

Typically, condition variables are used in constructs similar to this pseudocode:

```
lock mutex;
if condition has occurred
   signal thread(s);
else {
   unlock the mutex and block;
   /* when thread is unblocked, mutex is relocked */
}
unlock mutex;
```

Condition variables in Pthreads have type pthread_cond_t. The function

```
int pthread_cond_signal(pthread_cond_t*  cond_var_p /* in/out */);
```

will unblock *one* of the blocked threads, and

```
int pthread_cond_broadcast(pthread_cond_t* cond_var_p /* in/out */);
```

will unblock *all* of the blocked threads. The function

```
int pthread_cond_wait(
      pthread_cond_t*    cond_var_p   /* in/out */,
      pthread_mutex_t*   mutex_p      /* in/out */);
```

will unlock the mutex referred to by `mutex_p` and cause the executing thread to block until it is unblocked by another thread's call to `pthread_cond_signal` or `pthread_cond_broadcast`. When the thread is unblocked, it reacquires the mutex. So in effect, `pthread_cond_wait` implements the following sequence of functions:

```
pthread_mutex_unlock(&mutex_p);
wait_on_signal(&cond_var_p);
pthread_mutex_lock(&mutex_p);
```

The following code implements a barrier with a condition variable:

```
/* Shared */
int counter = 0;
pthread_mutex_t mutex;
pthread_cond_t cond_var;
. . .
void* Thread_work(. . .) {
   . . .
   /* Barrier */
   pthread_mutex_lock(&mutex);
   counter++;
   if (counter == thread_count) {
      counter = 0;
      pthread_cond_broadcast(&cond_var);
   } else {
      while (pthread_cond_wait(&cond_var, &mutex) != 0);
   }
   pthread_mutex_unlock(&mutex);
   . . .
}
```

Note that it is possible that events other than the call to `pthread_cond_broadcast` can cause a suspended thread to unblock (see, for example, Butenhof [6], page 80). Hence, the call to `pthread_cond_wait` is usually placed in a `while` loop. If the thread is unblocked by some event other than a call to `pthread_cond_signal` or `pthread_cond_broadcast`, then the return value of `pthread_cond_wait` will be nonzero, and the unblocked thread will call `pthread_cond_wait` again.

If a single thread is being awakened, it's also a good idea to check that the condition has, in fact, been satisfied before proceeding. In our example, if a single thread were being released from the barrier with a call to `pthread_cond_signal`, then that thread should verify that `counter == 0` before proceeding. This can be dangerous with the broadcast, though. After being awakened, some thread may race ahead and change the condition, and if each thread is checking the condition,

a thread that awakened later may find the condition is no longer satisfied and go back to sleep.

Note that in order for our barrier to function correctly, it's essential that the call to `pthread_cond_wait` unlock the mutex. If it didn't unlock the mutex, then only one thread could enter the barrier; all of the other threads would block in the call to `pthread_mutex_lock`, the first thread to enter the barrier would block in the call to `pthread_cond_wait`, and our program would hang.

Also note that the semantics of mutexes requires that the mutex be relocked before we return from the call to `pthread_cond_wait`. We "obtained" the lock when we returned from the call to `pthread_mutex_lock`. Hence, we should at some point "relinquish" the lock through a call to `pthread_mutex_unlock`.

Like mutexes and semaphores, condition variables should be initialized and destroyed. In this case, the functions are

```
int pthread_cond_init(
      pthread_cond_t*            cond_p       /* out */,
      const pthread_condattr_t*  cond_attr_p  /* in  */);

int pthread_cond_destroy(pthread_cond_t*  cond_p  /* in/out */);
```

We won't be using the second argument to `pthread_cond_init` (we'll call it with second argument `NULL`).

4.8.4 Pthreads barriers

Before proceeding we should note that the Open Group, the standards group that is continuing to develop the POSIX standard, does define a barrier interface for Pthreads. However, as we noted earlier, it is not universally available, so we haven't discussed it in the text. See Exercise 4.9 for some of the details of the API.

4.9 READ-WRITE LOCKS

Let's take a look at the problem of controlling access to a large, shared data structure, which can be either simply searched or updated by the threads. For the sake of explicitness, let's suppose the shared data structure is a sorted linked list of `int`s, and the operations of interest are `Member`, `Insert`, and `Delete`.

4.9.1 Linked list functions

The list itself is composed of a collection of list *nodes*, each of which is a struct with two members: an `int` and a pointer to the next node. We can define such a struct with the definition

```
struct list_node_s {
   int data;
   struct list_node_s* next;
}
```

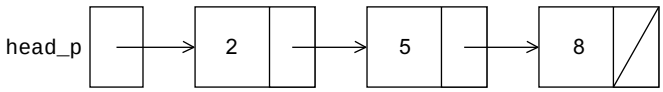


FIGURE 4.4

A linked list

```
1  int  Member(int value, struct list_node_s* head_p) {
2     struct list_node_s* curr_p = head_p;
3
4     while (curr_p != NULL && curr_p->data < value)
5        curr_p = curr_p->next;
6
7     if (curr_p == NULL || curr_p->data > value) {
8        return 0;
9     } else {
10       return 1;
11    }
12 }  /* Member */
```

Program 4.9: The `Member` function

A typical list is shown in Figure 4.4. A pointer, `head_p`, with type `struct list_node_s*` refers to the first node in the list. The `next` member of the last node is `NULL` (which is indicated by a slash (/) in the `next` member).

The `Member` function (Program 4.9) uses a pointer to traverse the list until it either finds the desired value or determines that the desired value cannot be in the list. Since the list is sorted, the latter condition occurs when the `curr_p` pointer is `NULL` or when the data member of the current node is larger than the desired value.

The `Insert` function (Program 4.10) begins by searching for the correct position in which to insert the new node. Since the list is sorted, it must search until it finds a node whose `data` member is greater than the `value` to be inserted. When it finds this node, it needs to insert the new node in the position *preceding* the node that's been found. Since the list is singly-linked, we can't "back up" to this position without traversing the list a second time. There are several approaches to dealing with this: the approach we use is to define a second pointer `pred_p`, which, in general, refers to the predecessor of the current node. When we exit the loop that searches for the position to insert, the `next` member of the node referred to by `pred_p` can be updated so that it refers to the new node. See Figure 4.5.

The `Delete` function (Program 4.11) is similar to the `Insert` function in that it also needs to keep track of the predecessor of the current node while it's searching for the node to be deleted. The predecessor node's `next` member can then be updated after the search is completed. See Figure 4.6.

```
1  int Insert(int value, struct list_node_s** head_p) {
2     struct list_node_s* curr_p = *head_p;
3     struct list_node_s* pred_p = NULL;
4     struct list_node_s* temp_p;
5
6     while (curr_p != NULL && curr_p->data < value) {
7        pred_p = curr_p;
8        curr_p = curr_p->next;
9     }
10
11    if (curr_p == NULL || curr_p->data > value) {
12       temp_p = malloc(sizeof(struct list_node_s));
13       temp_p->data = value;
14       temp_p->next = curr_p;
15       if (pred_p == NULL)  /* New first node */
16          *head_p = temp_p;
17       else
18          pred_p->next = temp_p;
19       return 1;
20    } else { /* Value already in list */
21       return 0;
22    }
23 }  /* Insert */
```

Program 4.10: The `Insert` function

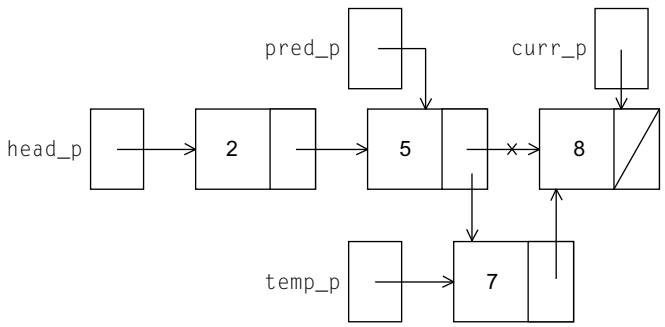


FIGURE 4.5

Inserting a new node into a list

4.9.2 A multithreaded linked list

Now let's try to use these functions in a Pthreads program. In order to share access to the list, we can define `head_p` to be a global variable. This will simplify the function headers for `Member`, `Insert`, and `Delete`, since we won't need to pass in either `head_p` or a pointer to `head_p`, we'll only need to pass in the value of interest.

```
1  int Delete(int value, struct list_node_s** head_p) {
2     struct list_node_s* curr_p = *head_p;
3     struct list_node_s* pred_p = NULL;
4
5     while (curr_p != NULL && curr_p->data < value) {
6        pred_p = curr_p;
7        curr_p = curr_p->next;
8     }
9
10    if (curr_p != NULL && curr_p->data == value) {
11       if (pred_p == NULL) { /* Deleting first node in list */
12          *head_p = curr_p->next;
13          free(curr_p);
14       } else {
15          pred_p->next = curr_p->next;
16          free(curr_p);
17       }
18       return 1;
19    } else {  /* Value isn't in list */
20       return 0;
21    }
22 }  /* Delete */
```

Program 4.11: The `Delete` function

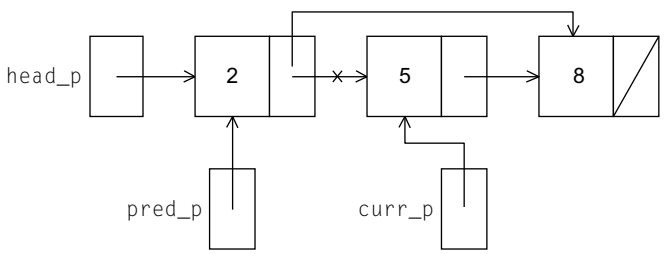


FIGURE 4.6

Deleting a node from the list

What now are the consequences of having multiple threads simultaneously execute the three functions?

Since multiple threads can simultaneously *read* a memory location without conflict, it should be clear that multiple threads can simultaneously execute `Member`. On the other hand, `Delete` and `Insert` also *write* to memory locations, so there may be problems if we try to execute either of these operations at the same time as another operation. As an example, suppose that thread 0 is executing `Member` (5) at the same time that thread 1 is executing `Delete` (5). The current state of the list is shown in Figure 4.7. An obvious problem is that if thread 0 is executing `Member` (5), it is going to report that 5 is in the list, when, in fact, it may be deleted even before thread 0

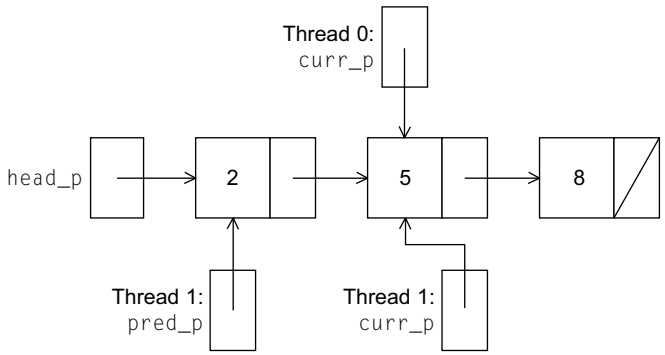


FIGURE 4.7

Simultaneous access by two threads

returns. A second obvious problem is if thread 0 is executing `Member` (8), thread 1 may free the memory used for the node storing 5 before thread 0 can advance to the node storing 8. Although typical implementations of `free` don't overwrite the freed memory, if the memory is reallocated before thread 0 advances, there can be serious problems. For example, if the memory is reallocated for use in something other than a list node, what thread 0 "thinks" is the `next` member may be set to utter garbage, and after it executes

```
curr_p = curr_p->next;
```

dereferencing `curr_p` may result in a segmentation violation.

More generally, we can run into problems if we try to simultaneously execute another operation while we're executing an `Insert` or a `Delete`. It's OK for multiple threads to simultaneously execute `Member`—that is, *read* the list nodes—but it's unsafe for multiple threads to access the list if at least one of the threads is executing an `Insert` or a `Delete`—that is, is *writing* to the list nodes (see Exercise 4.11).

How can we deal with this problem? An obvious solution is to simply lock the list any time that a thread attempts to access it. For example, a call to each of the three functions can be protected by a mutex, so we might execute

```
Pthread_mutex_lock(&list_mutex);
Member(value);
Pthread_mutex_unlock(&list_mutex);
```

instead of simply calling `Member(value)`.

An equally obvious problem with this solution is that we are serializing access to the list, and if the vast majority of our operations are calls to `Member`, we'll fail to exploit this opportunity for parallelism. On the other hand, if most of our operations are calls to `Insert` and `Delete`, then this may be the best solution, since we'll need to serialize access to the list for most of the operations, and this solution will certainly be easy to implement.

An alternative to this approach involves "finer-grained" locking. Instead of locking the entire list, we could try to lock individual nodes. We would add, for example, a mutex to the list node struct:

```
struct list_node_s {
   int data;
   struct list_node_s* next;
   pthread_mutex_t mutex;
}
```

Now each time we try to access a node we must first lock the mutex associated with the node. Note that this will also require that we have a mutex associated with the `head_p` pointer. So, for example, we might implement `Member` as shown in Program 4.12. Admittedly this implementation is *much* more complex than the original `Member` function. It is also much slower, since, in general, each time a node is accessed, a mutex must be locked and unlocked. At a minimum it will add two function calls to the node access, but it can also add a substantial delay if a thread has

```
int  Member(int value) {
   struct list_node_s* temp_p;

   pthread_mutex_lock(&head_p_mutex);
   temp_p = head_p;
   while (temp_p != NULL && temp_p->data < value) {
      if (temp_p->next != NULL)
         pthread_mutex_lock(&(temp_p->next->mutex));
      if (temp_p == head_p)
         pthread_mutex_unlock(&head_p_mutex);
      pthread_mutex_unlock(&(temp_p->mutex));
      temp_p = temp_p->next;
   }

   if (temp_p == NULL || temp_p->data > value) {
      if (temp_p == head_p)
         pthread_mutex_unlock(&head_p_mutex);
      if (temp_p != NULL)
         pthread_mutex_unlock(&(temp_p->mutex));
      return 0;
   } else {
      if (temp_p == head_p)
         pthread_mutex_unlock(&head_p_mutex);
      pthread_mutex_unlock(&(temp_p->mutex));
      return 1;
   }
}  /* Member */
```

Program 4.12: Implementation of `Member` with one mutex per list node

to wait for a lock. A further problem is that the addition of a mutex field to each node will substantially increase the amount of storage needed for the list. On the other hand, the finer-grained locking might be a closer approximation to what we want. Since we're only locking the nodes of current interest, multiple threads can simultaneously access different parts of the list, regardless of which operations they're executing.

4.9.3 Pthreads read-write locks

Neither of our multithreaded linked lists exploits the potential for simultaneous access to *any* node by threads that are executing `Member`. The first solution only allows one thread to access the entire list at any instant, and the second only allows one thread to access any given node at any instant. An alternative is provided by Pthreads' **read-write locks**. A read-write lock is somewhat like a mutex except that it provides two lock functions. The first lock function locks the read-write lock for *reading,* while the second locks it for *writing.* Multiple threads can thereby simultaneously obtain the lock by calling the read-lock function, while only one thread can obtain the lock by calling the write-lock function. Thus, if any threads own the lock for reading, any threads that want to obtain the lock for writing will block in the call to the write-lock function. Furthermore, if any thread owns the lock for writing, any threads that want to obtain the lock for reading or writing will block in their respective locking functions.

Using Pthreads read-write locks, we can protect our linked list functions with the following code (we're ignoring function return values):

```
pthread_rwlock_rdlock(&rwlock);
Member(value);
pthread_rwlock_unlock(&rwlock);
. . .
pthread_rwlock_wrlock(&rwlock);
Insert(value);
pthread_rwlock_unlock(&rwlock);
. . .
pthread_rwlock_wrlock(&rwlock);
Delete(value);
pthread_rwlock_unlock(&rwlock);
```

The syntax for the new Pthreads functions is

```
int pthread_rwlock_rdlock(pthread_rwlock_t*  rwlock_p /* in/out */);
int pthread_rwlock_wrlock(pthread_rwlock_t*  rwlock_p /* in/out */);
int pthread_rwlock_unlock(pthread_rwlock_t*  rwlock_p /* in/out */);
```

As their names suggest, the first function locks the read-write lock for reading, the second locks it for writing, and the last unlocks it.

As with mutexes, read-write locks should be initialized before use and destroyed after use. The following function can be used for initialization:

```
int pthread_rwlock_init(
      pthread_rwlock_t*            rwlock_p  /* out */,
      const pthread_rwlockattr_t*  attr_p    /* in  */);
```

Also as with mutexes, we'll not use the second argument, so we'll just pass `NULL`. The following function can be used for destruction of a read-write lock:

```
int pthread_rwlock_destroy(pthread_rwlock_t* rwlock_p /* in/out */);
```

4.9.4 Performance of the various implementations

Of course, we really want to know which of the three implementations is "best," so we included our implementations in a small program in which the main thread first inserts a user-specified number of randomly generated keys into an empty list. After being started by the main thread, each thread carries out a user-specified number of operations on the list. The user also specifies the percentages of each type of operation (`Member`, `Insert`, `Delete`). However, which operation occurs when and on which key is determined by a random number generator. Thus, for example, the user might specify that 1000 keys should be inserted into an initially empty list and a total of 100,000 operations are to be carried out by the threads. Further, she might specify that 80% of the operations should be `Member`, 15% should be `Insert`, and the remaining 5% should be `Delete`. However, since the operations are randomly generated, it might happen that the threads execute a total of, say, 79,000 calls to `Member`, 15,500 calls to `Insert`, and 5500 calls to `Delete`.

Tables 4.3 and 4.4 show the times (in seconds) that it took for 100,000 operations on a list that was initialized to contain 1000 keys. Both sets of data were taken on a system containing four dual-core processors.

Table 4.3 shows the times when 99.9% of the operations are `Member` and the remaining 0.1% are divided equally between `Insert` and `Delete`. Table 4.4 shows the times when 80% of the operations are `Member`, 10% are `Insert`, and 10% are `Delete`. Note that in both tables when one thread is used, the run-times for the

Table 4.3 Linked List Times: 1000 Initial Keys, 100,000 ops, 99.9% `Member`, 0.05% `Insert`, 0.05% `Delete`

	Number of Threads			
Implementation	**1**	**2**	**4**	**8**
Read-Write Locks	0.213	0.123	0.098	0.115
One Mutex for Entire List	0.211	0.450	0.385	0.457
One Mutex per Node	1.680	5.700	3.450	2.700

Table 4.4 Linked List Times: 1000 Initial Keys, 100,000 ops, 80% `Member`, 10% `Insert`, 10% `Delete`

	Number of Threads			
Implementation	**1**	**2**	**4**	**8**
Read-Write Locks	2.48	4.97	4.69	4.71
One Mutex for Entire List	2.50	5.13	5.04	5.11
One Mutex per Node	12.00	29.60	17.00	12.00

read-write locks and the single-mutex implementations are about the same. This makes sense: the operations are serialized, and since there is no contention for the read-write lock or the mutex, the overhead associated with both implementations should consist of a function call before the list operation and a function call after the operation. On the other hand, the implementation that uses one mutex per node is *much* slower. This also makes sense, since each time a single node is accessed there will be two function calls—one to lock the node mutex and one to unlock it. Thus, there's considerably more overhead for this implementation.

The inferiority of the implementation that uses one mutex per node persists when we use multiple threads. There is far too much overhead associated with all the locking and unlocking to make this implementation competitive with the other two implementations.

Perhaps the most striking difference between the two tables is the relative performance of the read-write lock implementation and the single-mutex implementation when multiple threads are used. When there are very few `Insert`s and `Delete`s, the read-write lock implementation is far better than the single-mutex implementation. Since the single-mutex implementation will serialize all the operations, this suggests that if there are very few `Insert`s and `Delete`s, the read-write locks do a very good job of allowing concurrent access to the list. On the other hand, if there are a relatively large number of `Insert`s and `Delete`s (for example, 10% each), there's very little difference between the performance of the read-write lock implementation and the single-mutex implementation. Thus, for linked list operations, read-write locks *can* provide a considerable increase in performance, but only if the number of `Insert`s and `Delete`s is quite small.

Also notice that if we use one mutex or one mutex per node, the program is *always* as fast or faster when it's run with one thread. Furthermore, when the number of `insert`s and `delete`s is relatively large, the read-write lock program is also faster with one thread. This isn't surprising for the one mutex implementation, since effectively accesses to the list are serialized. For the read-write lock implementation, it appears that when there are a substantial number of write-locks, there is too much contention for the locks and overall performance deteriorates significantly.

In summary, the read-write lock implementation is superior to the one mutex and the one mutex per node implementations. However, unless the number of `insert`s and `delete`s is small, a serial implementation will be superior.

4.9.5 Implementing read-write locks

The original Pthreads specification didn't include read-write locks, so some of the early texts describing Pthreads include implementations of read-write locks (see, for example, [6]). A typical implementation[6] defines a data structure that uses two condition variables—one for "readers" and one for "writers"—and a mutex. The structure also contains members that indicate

1. how many readers own the lock, that is, are currently reading,
2. how many readers are waiting to obtain the lock,
3. whether a writer owns the lock, and
4. how many writers are waiting to obtain the lock.

The mutex protects the read-write lock data structure: whenever a thread calls one of the functions (read-lock, write-lock, unlock), it first locks the mutex, and whenever a thread completes one of these calls, it unlocks the mutex. After acquiring the mutex, the thread checks the appropriate data members to determine how to proceed. As an example, if it wants read-access, it can check to see if there's a writer that currently owns the lock. If not, it increments the number of active readers and proceeds. If a writer is active, it increments the number of readers waiting and starts a condition wait on the reader condition variable. When it's awakened, it decrements the number of readers waiting, increments the number of active readers, and proceeds. The write-lock function has an implementation that's similar to the read-lock function.

The action taken in the unlock function depends on whether the thread was a reader or a writer. If the thread was a reader, there are no currently active readers, *and* there's a writer waiting, then it can signal a writer to proceed before returning. If, on the other hand, the thread was a writer, there can be both readers and writers waiting, so the thread needs to decide whether it will give preference to readers or writers. Since writers must have exclusive access, it is likely that it is much more difficult for a writer to obtain the lock. Many implementations therefore give writers preference. Programming Assignment 4.6 explores this further.

4.10 CACHES, CACHE COHERENCE, AND FALSE SHARING[7]

Recall that for a number of years now, processors have been able to execute operations much faster than they can access data in main memory. If a processor must

[6] This discussion follows the basic outline of Butenhof's implementation [6].

[7] This material is also covered in Chapter 5, so if you've already read that chapter, you may want to skim this section.

read data from main memory for each operation, it will spend most of its time simply waiting for the data from memory to arrive. Also recall that in order to address this problem, chip designers have added blocks of relatively fast memory to processors. This faster memory is called **cache memory**.

The design of cache memory takes into consideration the principles of **temporal and spatial locality**: if a processor accesses main memory location x at time t, then it is likely that at times close to t it will access main memory locations close to x. Thus, if a processor needs to access main memory location x, rather than transferring only the contents of x to/from main memory, a block of memory containing x is tranferred from/to the processor's cache. Such a block of memory is called a **cache line** or **cache block**.

In Section 2.3.4, we saw that the use of cache memory can have a huge impact on shared-memory. Let's recall why. First, consider the following situation: Suppose `x` is a shared variable with the value five, and both thread 0 and thread 1 read `x` from memory into their (separate) caches, because both want to execute the statement

```
my_y = x;
```

Here, `my_y` is a private variable defined by both threads. Now suppose thread 0 executes the statement

```
x++;
```

Finally, suppose that thread 1 now executes

```
my_z = x;
```

where `my_z` is another private variable.

What's the value in `my_z`? Is it five? Or is it six? The problem is that there are (at least) three copies of `x`: the one in main memory, the one in thread 0's cache, and the one in thread 1's cache. When thread 0 executed `x++`, what happened to the values in main memory and thread 1's cache? This is the **cache coherence** problem, which we discussed in Chapter 2. We saw there that most systems insist that the caches be made aware that changes have been made to data they are caching. The line in the cache of thread 1 would have been marked *invalid* when thread 0 executed `x++`, and before assigning `my_z = x`, the core running thread 1 would see that its value of `x` was out of date. Thus, the core running thread 0 would have to update the copy of `x` in main memory (either now or earlier), and the core running thread 1 would get the line with the updated value of `x` from main memory. For further details, see Chapter 2.

The use of cache coherence can have a dramatic effect on the performance of shared-memory systems. To illustrate this, recall our Pthreads matrix-vector multiplication example: The main thread initialized an $m \times n$ matrix A and an n-dimensional vector $\mathbf{x}$. Each thread was responsible for computing m/t components of the product vector $\mathbf{y} = A\mathbf{x}$. (As usual, t is the number of threads.) The data structures representing A, $\mathbf{x}$, $\mathbf{y}$, m, and n were all shared. For ease of reference, we reproduce the code in Program 4.13.

If T_{serial} is the run-time of the serial program and T_{parallel} is the run-time of the parallel program, recall that the *efficiency* E of the parallel program is the speedup S divided by the number of threads:

$$E = \frac{S}{t} = \frac{\left(\frac{T_{\text{serial}}}{T_{\text{parallel}}}\right)}{t} = \frac{T_{\text{serial}}}{t \times T_{\text{parallel}}}.$$

Since $S \leq t$, $E \leq 1$. Table 4.5 shows the run-times and efficiencies of our matrix-vector multiplication with different sets of data and differing numbers of threads.

In each case, the total number of floating point additions and multiplications is $64,000,000$, so an analysis that only considers arithmetic operations would predict that a single thread running the code would take the same amount of time for all three inputs. However, it's clear that this is *not* the case. With one thread, the $8,000,000 \times 8$ system requires about 14% more time than the 8000×8000 system, and the $8 \times 8,000,000$ system requires about 28% more time than the 8000×8000 system. Both of these differences are at least partially attributable to cache performance.

```
1  void *Pth_mat_vect(void* rank) {
2     long my_rank = (long) rank;
3     int i, j;
4     int local_m = m/thread_count;
5     int my_first_row = my_rank*local_m;
6     int my_last_row = (my_rank+1)*local_m - 1;
7
8     for (i = my_first_row; i <= my_last_row; i++) {
9        y[i] = 0.0;
10       for (j = 0; j < n; j++)
11           y[i] += A[i][j]*x[j];
12    }
13
14    return NULL;
15 }  /* Pth_mat_vect */
```

Program 4.13: Pthreads matrix-vector multiplication

Table 4.5 Run-Times and Efficiencies of Matrix-Vector Multiplication (times are in seconds)

	Matrix Dimension					
	$8,000,000 \times 8$		8000×8000		$8 \times 8,000,000$	
Threads	Time	Eff.	Time	Eff.	Time	Eff.
1	0.393	1.000	0.345	1.000	0.441	1.000
2	0.217	0.906	0.188	0.918	0.300	0.735
4	0.139	0.707	0.115	0.750	0.388	0.290

Recall that a *write-miss* occurs when a core tries to update a variable that's not in the cache, and it has to access main memory. A cache profiler (such as Valgrind [49]) shows that when the program is run with the $8,000,000 \times 8$ input, it has far more cache write-misses than either of the other inputs. The bulk of these occur in Line 9. Since the number of elements in the vector `y` is far greater in this case (8,000,000 vs. 8000 or 8), and each element must be initialized, it's not surprising that this line slows down the execution of the program with the $8,000,000 \times 8$ input.

Also recall that a *read-miss* occurs when a core tries to read a variable that's not in the cache, and it has to access main memory. A cache profiler shows that when the program is run with the $8 \times 8,000,000$ input, it has far more cache read-misses than either of the other inputs. These occur in Line 11, and a careful study of this program (see Exercise 4.15) shows that the main source of the differences is due to the reads of `x`. Once again, this isn't surprising, since for this input, `x` has 8,000,000 elements, versus only 8000 or 8 for the other inputs.

It should be noted that there may be other factors that are affecting the relative performance of the single-threaded program with the differing inputs. For example, we haven't taken into consideration whether virtual memory (see Section 2.2.4) has affected the performance of the program with the different inputs. How frequently does the CPU need to access the page table in main memory?

Of more interest to us, though, is the tremendous difference in efficiency as the number of threads is increased. The two-thread efficiency of the program with the $8 \times 8,000,000$ input is nearly 20% less than the efficiency of the program with the $8,000,000 \times 8$ and the 8000×8000 inputs. The four-thread efficiency of the program with the $8 \times 8,000,000$ input is nearly 60% less than the program's efficiency with the $8,000,000 \times 8$ input and *more* than 60% less than the program's efficiency with the 8000×8000 input. These dramatic decreases in efficiency are even more remarkable when we note that with one thread the program is much slower with $8 \times 8,000,000$ input. Therefore, the numerator in the formula for the efficiency:

$$\text{Parallel Efficiency} = \frac{\text{Serial Run-Time}}{(\text{Number of Threads}) \times (\text{Parallel Run-Time})}$$

will be much larger. Why, then, is the multithreaded performance of the program so much worse with the $8 \times 8,000,000$ input?

In this case, once again, the answer has to do with cache. Let's take a look at the program when we run it with four threads. With the $8,000,000 \times 8$ input, `y` has 8,000,000 components, so each thread is assigned 2,000,000 components. With the 8000×8000 input, each thread is assigned 2000 components of `y`, and with the $8 \times 8,000,000$ input, each thread is assigned 2 components. On the system we used, a cache line is 64 bytes. Since the type of `y` is `double`, and a `double` is 8 bytes, a single cache line can store 8 `doubles`.

Cache coherence is enforced at the "cache-line level." That is, each time any value in a cache line is written, if the line is also stored in another processor's cache, the entire *line* will be invalidated—not just the value that was written. The system we're using has two dual-core processors and each processor has its own cache. Suppose

for the moment that threads 0 and 1 are assigned to one of the processors and threads 2 and 3 are assigned to the other. Also suppose that for the $8 \times 8{,}000{,}000$ problem all of `y` is stored in a single cache line. Then every write to some element of `y` will invalidate the line in the other processor's cache. For example, each time thread 0 updates `y[0]` in the statement

```
y[i] += A[i][j]*x[j];
```

If thread 2 or 3 is executing this code, it will have to reload `y`. Each thread will update each of its components $8{,}000{,}000$ times. We see that with this assignment of threads to processors and components of `y` to cache lines, all the threads will have to reload `y` *many* times. This is going to happen in spite of the fact that only one thread accesses any one component of `y`—for example, only thread 0 accesses `y[0]`.

Each thread will update its assigned components of `y` a total of 16,000,000 times. It appears that many, if not most, of these updates are forcing the threads to access main memory. This is called **false sharing**. Suppose two threads with separate caches access different variables that belong to the same cache line. Further suppose at least one of the threads updates its variable. Then even though neither thread has written to a variable that the other thread is using, the cache controller invalidates the entire cache line and forces the threads to get the values of the variables from main memory. The threads aren't sharing anything (except a cache line), but the behavior of the threads with respect to memory access is the same as if they were sharing a variable, hence the name *false sharing*.

Why is false sharing not a problem with the other inputs? Let's look at what happens with the 8000×8000 input. Suppose thread 2 is assigned to one of the processors and thread 3 is assigned to another. (We don't actually know which threads are assigned to which processors, but it turns out—see Exercise 4.16—that it doesn't matter.) Thread 2 is responsiblefor computing

```
y[4000], y[4001], . . . , y[5999],
```

and thread 3 is responsible for computing

```
y[6000], y[6001], . . . , y[7999].
```

If a cache line contains 8 consecutive `doubles`, the only possibility for false sharing is on the interface between their assigned elements. If, for example, a single cache line contains

```
y[5996], y[5997], y[5998], y[5999], y[6000], y[6001], y[6002], y[6003],
```

then it's conceivable that there might be false sharing of this cache line. However, thread 2 will access

```
y[5996], y[5997], y[5998], y[5999]
```

at the *end* of its `for i` loop, while thread 3 will access

```
y[6000], y[6001], y[6002], y[6003]
```

at the *beginning* of its `for i` loop. So it's very likely that when thread 2 accesses (say) `y[5996]`, thread 3 will be long done with all four of

```
y[6000], y[6001], y[6002], y[6003].
```

Similarly, when thread 3 accesses, say, `y[6003]`, it's very likely that thread 2 won't be anywhere near starting to access

```
y[5996], y[5997], y[5998], y[5999].
```

It's therefore unlikely that false sharing of the elements of `y` will be a significant problem with the 8000×8000 input. Similar reasoning suggests that false sharing of `y` is unlikely to be a problem with the $8,000,000 \times 8$ input. Also note that we don't need to worry about false sharing of `A` or `x`, since their values are never updated by the matrix-vector multiplication code.

This brings up the question of how we might avoid false sharing in our matrix-vector multiplication program. One possible solution is to "pad" the `y` vector with dummy elements in order to insure that any update by one thread won't affect another thread's cache line. Another alternative is to have each thread use its own private storage during the multiplication loop, and then update the shared storage when they're done. See Exercise 4.18.

4.11 THREAD-SAFETY[8]

Let's look at another potential problem that occurs in shared-memory programming: *thread-safety.* A block of code is **thread-safe** if it can be simultaneously executed by multiple threads without causing problems.

As an example, suppose we want to use multiple threads to "tokenize" a file. Let's suppose that the file consists of ordinary English text, and that the tokens are just contiguous sequences of characters separated from the rest of the text by white space—a space, a tab, or a newline. A simple approach to this problem is to divide the input file into lines of text and assign the lines to the threads in a round-robin fashion: the first line goes to thread 0, the second goes to thread 1, ..., the tth goes to thread t, the $t+1$st goes to thread 0, and so on.

We can serialize access to the lines of input using semaphores. Then, after a thread has read a single line of input, it can tokenize the line. One way to do this is to use the `strtok` function in `string.h`, which has the following prototype:

```
char* strtok(
      char*        string      /* in/out */,
      const char*  separators  /* in     */);
```

Its usage is a little unusual: the first time it's called the `string` argument should be the text to be tokenized, so in our example it should be the line of input. For subsequent

[8]This material is also covered in Chapter 5, so if you've already read that chapter, you may want to skim this section.

calls, the first argument should be `NULL`. The idea is that in the first call, `strtok` caches a pointer to `string`, and for subsequent calls it returns successive tokens taken from the cached copy. The characters that delimit tokens should be passed in `separators`. We should pass in the string `" \t\n"` as the `separators` argument.

Given these assumptions, we can write the thread function shown in Program 4.14. The main thread has initialized an array of t semaphores—one for each thread. Thread 0's semaphore is initialized to 1. All the other semaphores are initialized to 0. So the code in Lines 9 to 11 will force the threads to sequentially access the lines of input. Thread 0 will immediately read the first line, but all the other threads will block in `sem_wait`. When thread 0 executes the `sem_post`, thread 1 can read a line of input. After each thread has read its first line of input (or end-of-file), any additional input is read in Lines 24 to 26. The `fgets` function reads a single line of input and Lines 15 to 22 identify the tokens in the line. When we run the

```
1  void* Tokenize(void* rank) {
2     long my_rank = (long) rank;
3     int count;
4     int next = (my_rank + 1) % thread_count;
5     char *fg_rv;
6     char my_line[MAX];
7     char *my_string;
8
9     sem_wait(&sems[my_rank]);
10    fg_rv = fgets(my_line, MAX, stdin);
11    sem_post(&sems[next]);
12    while (fg_rv != NULL) {
13       printf("Thread %ld > my line = %s", my_rank, my_line);
14
15       count = 0;
16       my_string = strtok(my_line, " \t\n");
17       while ( my_string != NULL ) {
18          count++;
19          printf("Thread %ld > string %d = %s\n", my_rank, count,
20                my_string);
21          my_string = strtok(NULL, " \t\n");
22       }
23
24       sem_wait(&sems[my_rank]);
25       fg_rv = fgets(my_line, MAX, stdin);
26       sem_post(&sems[next]);
27    }
28
29    return NULL;
30 }  /* Tokenize */
```

Program 4.14: A first attempt at a multithreaded tokenizer

program with a single thread, it correctly tokenizes the input stream. The first time we run it with two threads and the input

Pease porridge hot.
Pease porridge cold.
Pease porridge in the pot
Nine days old.

the output is also correct. However, the second time we run it with this input, we get the following output.

```
Thread 0 > my line = Pease porridge hot.
Thread 0 > string 1 = Pease
Thread 0 > string 2 = porridge
Thread 0 > string 3 = hot.
Thread 1 > my line = Pease porridge cold.
Thread 0 > my line = Pease porridge in the pot
Thread 0 > string 1 = Pease
Thread 0 > string 2 = porridge
Thread 0 > string 3 = in
Thread 0 > string 4 = the
Thread 0 > string 5 = pot
Thread 1 > string 1 = Pease
Thread 1 > my line = Nine days old.
Thread 1 > string 1 = Nine
Thread 1 > string 2 = days
Thread 1 > string 3 = old.
```

What happened? Recall that `strtok` caches the input line. It does this by declaring a variable to have `static` storage class. This causes the value stored in this variable to persist from one call to the next. Unfortunately for us, this cached string is shared, not private. Thus, thread 0's call to `strtok` with the third line of the input has apparently overwritten the contents of thread 1's call with the second line.

The `strtok` function is *not* thread-safe: if multiple threads call it simultaneously, the output it produces may not be correct. Regrettably, it's not uncommon for C library functions to fail to be thread-safe. For example, neither the random number generator `random` in `stdlib.h` nor the time conversion function `localtime` in `time.h` is thread-safe. In some cases, the C standard specifies an alternate, thread-safe version of a function. In fact, there is a thread-safe version of `strtok`:

```
char* strtok_r(
      char*        string       /* in/out */,
      const char*  separators   /* in     */,
      char**       saveptr_p    /* in/out */);
```

The "`_r`" is supposed to suggest that the function is *reentrant,* which is sometimes used as a synonym for thread-safe. The first two arguments have the same purpose as the arguments to `strtok`. The `saveptr Append ''_p'' to ''saveptr''` argument is used by `strtok_r` for keeping track of where the function is in the input string; it serves the purpose of the cached pointer in `strtok`. We can correct our original

Tokenize function by replacing the calls to strtok with calls to strtok_r. We simply need to declare a char* variable to pass in for the third argument, and replace the calls in Line 16 and Line 21 with the calls

```
my_string = strtok_r(my_line, " \t\n", &saveptr);
. . .
my_string = strtok_r(NULL, " \t\n", &saveptr);
```

respectively.

4.11.1 Incorrect programs can produce correct output

Notice that our original version of the tokenizer program shows an especially insidious form of program error: the first time we ran it with two threads, the program produced correct output. It wasn't until a later run that we saw an error. This, unfortunately, is not a rare occurrence in parallel programs. It's especially common in shared-memory programs. Since, for the most part, the threads are running independently of each other, as we noted earlier, the exact sequence of statements executed is nondeterministic. For example, we can't say when thread 1 will first call strtok. If its first call takes place after thread 0 has tokenized its first line, then the tokens identified for the first line should be correct. However, if thread 1 calls strtok before thread 0 has finished tokenizing its first line, it's entirely possible that thread 0 may not identify all the tokens in the first line. Therefore, it's especially important in developing shared-memory programs to resist the temptation to assume that since a program produces correct output, it must be correct. We always need to be wary of race conditions.

4.12 SUMMARY

Like MPI, Pthreads is a library of functions that programmers can use to implement parallel programs. Unlike MPI, Pthreads is used to implement shared-memory parallelism.

A **thread** in shared-memory programming is analogous to a process that is in distributed-memory programming. However, a thread is often lighter-weight than a full-fledged process.

We saw that in Pthreads programs, all the threads have access to global variables, while local variables usually are private to the thread running the function. In order to use Pthreads, we should include the pthread.h header file, and, when we compile our program, it may be necessary to link our program with the Pthread library by adding -lpthread to the command line. We saw that we can use the functions pthread_create and pthread_join, respectively, to start and stop a thread function.

When multiple threads are executing, the order in which the statements are executed by the different threads is usually nondeterministic. When nondeterminism results from multiple threads attempting to access a shared resource such as a shared variable or a shared file, at least one of the accesses is an update, and the accesses

can result in an error, we have a **race condition**. One of our most important tasks in writing shared-memory programs is identifying and correcting race conditions. A **critical section** is a block of code that updates a shared resource that can only be updated by one thread at a time, so the execution of code in a critical section should, effectively, be executed as serial code. Thus, we should try to design our programs so that they use them as infrequently as possible, and the critical sections we do use should be as short as possible.

We looked at three basic approaches to avoiding conflicting access to critical sections: busy-waiting, mutexes, and semaphores. **Busy-waiting** can be done with a flag variable and a `while` loop with an empty body. It can be very wasteful of CPU cycles. It can also be unreliable if compiler optimization is turned on, so mutexes and semaphores are generally preferable.

A **mutex** can be thought of as a lock on a critical section, since mutexes arrange for *mutually exclusive* access to a critical section. In Pthreads, a thread attempts to obtain a mutex with a call to `pthread_mutex_lock`, and it relinquishes the mutex with a call to `pthread_mutex_unlock`. When a thread attempts to obtain a mutex that is already in use, it *blocks* in the call to `pthread_mutex_lock`. This means that it remains idle in the call to `pthread_mutex_lock` until the system gives it the lock. A **semaphore** is an **unsigned** `int` together with two operations: `sem_wait` and `sem_post`. If the semaphore is positive, a call to `sem_wait` simply decrements the semaphore, but if the semaphore is zero, the calling thread blocks until the semaphore is positive, at which point the semaphore is decremented and the thread returns from the call. The `sem_post` operation increments the semaphore, so a semaphore can be used as a mutex with `sem_wait` corresponding to `pthread_mutex_lock` and `sem_post` corresponding to `pthread_mutex_unlock`. However, semaphores are more powerful than mutexes since they can be initialized to any nonnegative value. Furthermore, since there is no "ownership" of a semaphore, any thread can "unlock" a locked semaphore. We saw that semaphores can be easily used to implement **producer-consumer synchronization**. In producer-consumer synchronization, a "consumer" thread waits for some condition or data created by a "producer" thread before proceeding. Semaphores are not part of Pthreads. In order to use them, we need to include the `semaphore.h` header file.

A **barrier** is a point in a program at which the threads block until all of the threads have reached it. We saw several different means for constructing barriers. One of them used a **condition variable**. A condition variable is a special Pthreads object that can be used to suspend execution of a thread until a condition has occurred. When the condition has occurred, another thread can awaken the suspended thread with a condition signal or a condition broadcast.

The last Pthreads construct we looked at was a **read-write lock**. A read-write lock is used when it's safe for multiple threads to simultaneously *read* a data structure, but if a thread needs to modify or *write* to the data structure, then only that thread can access the data structure during the modification.

We recalled that modern microprocessor architectures use caches to reduce memory access times, so typical architectures have special hardware to insure that the caches on the different chips are **coherent**. Since the unit of cache coherence, a **cache**

line or **cache block**, is usually larger than a single word of memory, this can have the unfortunate side effect that two threads may be accessing different memory locations, but when the two locations belong to the same cache line, the cache-coherence hardware acts as if the threads were accessing the same memory location. Thus, if one of the threads updates its memory location, and then the other thread tries to read its memory location, it will have to retrieve the value from main memory. That is, the hardware is forcing the thread to act as if it were actually sharing the memory location. Hence, this is called **false sharing**, and it can seriously degrade the performance of a shared-memory program.

Some C functions cache data between calls by declaring variables to be `static`. This can cause errors when multiple threads call the function; since static storage is shared among the threads, one thread can overwrite another thread's data. Such a function is not **thread-safe**, and, unfortunately, there are serval such functions in the C library. Sometimes, however, there is a thread-safe variant.

When we looked at the program that used the function that wasn't thread-safe, we saw a particularly insidious problem: when we ran the program with multiple threads and a fixed set of input, it sometimes produced correct output, even though the program was erroneous. This means that even if a program produces correct output during testing, there's no guarantee that it is in fact correct–it's up to us to identify possible race condtions.

4.13 EXERCISES

4.1. When we discussed matrix-vector multiplication we assumed that both m and n, the number of rows and the number of columns, respectively, were evenly divisible by t, the number of threads. How do the formulas for the assignments change if this is *not* the case?

4.2. If we decide to physically divide a data structure among the threads, that is, if we decide to make various members local to individual threads, we need to consider at least three issues:

a. How are the members of the data structure used by the individual threads?
b. Where and how is the data structure initialized?
c. Where and how is the data structure used after its members are computed?

We briefly looked at the first issue in the matrix-vector multiplication function. We saw that the entire vector `x` was used by all of the threads, so it seemed pretty clear that it should be shared. However, for both the matrix `A` and the product vector `y`, just looking at (a) seemed to suggest that `A` and `y` should have their components distributed among the threads. Let's take a closer look at this.

What would we have to do in order to divide `A` and `y` among the threads? Dividing `y` wouldn't be difficult–each thread could allocate a block of memory that could be used for storing its assigned components. Presumably, we could do the same for `A`–each thread could allocate a block of memory for storing

its assigned rows. Modify the matrix-vector multiplication program so that it distributes both of these data structures. Can you "schedule" the input and output so that the threads can read in `A` and print out `y`? How does distributing `A` and `y` affect the run-time of the matrix-vector multiplication? (Don't include input or output in your run-time.)

4.3. Recall that the compiler is unaware that an ordinary C program is multithreaded, and as a consequence, it may make optimizations that can interfere with busy-waiting. (Note that compiler optimizations should *not* affect mutexes, condition variables, or semaphores.) An alternative to completely turning off compiler optimizations is to identify some shared variables with the C keyword **volatile**. This tells the compiler that these variables may be updated by mutiple threads and, as a consequence, it shouldn't apply optimizations to statements involving them. As an example, recall our busy-wait solution to the race condition when multiple threads attempt to add a private variable into a shared variable:

```
/* x and flag are shared, y is private            */
/* x and flag are initialized to 0 by main thread */

y = Compute(my_rank);
while (flag != my_rank);
x = x + y;
flag++;
```

It's impossible to tell by looking at this code that the order of the `while` statement and the `x = x + y` statement is important; if this code were single-threaded, the order of these two statements wouldn't affect the outcome of the code. But if the compiler determined that it could improve register usage by interchanging the order of these two statements, the resulting code would be erroneous.

If, instead of defining

```
int flag;
int x;
```

we define

```
int volatile flag;
int volatile x;
```

then the compiler will know that both `x` and `flag` can be updated by other threads, so it shouldn't try reordering the statements.

With the `gcc` compiler, the default behavior is no optimization. You can make certain of this by adding the option `-O0` to the command line. Try running the π calculation program that uses busy-waiting (`pth_pi_busy.c`) without optimization. How does the result of the multithreaded calculation compare to the single-threaded calculation? Now try running it with optimization; if you're using `gcc`, replace the `-O0` option with `-O2`. If you found an error, how many threads did you use?

Which variables should be made volatile in the π calculation? Change these variables so that they're volatile and rerun the program with and without optimization. How do the results compare to the single-threaded program?

4.4. The performance of the π calculation program that uses mutexes remains roughly constant once we increase the number of threads beyond the number of available CPUs. What does this suggest about how the threads are scheduled on the available processors?

4.5. Modify the mutex version of the π calculation program so that the critical section is in the `for` loop. How does the performance of this version compare to the performance of the original busy-wait version? How might we explain this?

4.6. Modify the mutex version of the π calculation program so that it uses a semaphore instead of a mutex. How does the performance of this version compare with the mutex version?

4.7. Although producer-consumer synchronization is easy to implement with semaphores, it's also possible to implement it with mutexes. The basic idea is to have the producer and the consumer share a mutex. A flag variable that's initialized to `false` by the main thread indicates whether there's anything to consume. With two threads we'd execute something like this:

```
while (1) {
   pthread_mutex_lock(&mutex);
   if (my_rank == consumer) {
      if (message_available) {
         print message;
         pthread_mutex_unlock(&mutex);
         break;
      }
   } else { /* my_rank == producer */
      create message;
      message_available = 1;
      pthread_mutex_unlock(&mutex);
      break;
   }
   pthread_mutex_unlock(&mutex);
}
```

So if the consumer gets into the loop first, it will see there's no message available and return to the call to `pthread_mutex_lock`. It will continue this process until the producer creates the message. Write a Pthreads program that implements this version of producer-consumer synchronization with two threads. Can you generalize this so that it works with 2k threads–odd-ranked threads are consumers and even-ranked threads are producers? Can you generalize this so that each thread is both a producer and a consumer? For example, suppose that thread q "sends" a message to thread $(q + 1)$ mod t and "receives" a message from thread $(q - 1 + t)$ mod t? Does this use busy-waiting?

4.8. If a program uses more than one mutex, and the mutexes can be acquired in different orders, the program can **deadlock**. That is, threads may block forever waiting to acquire one of the mutexes. As an example, suppose that a program has two shared data structures–for example, two arrays or two linked lists–each of which has an associated mutex. Further suppose that each data structure can be accessed (read or modified) after acquiring the data structure's associated mutex.

a. Suppose the program is run with two threads. Further suppose that the following sequence of events occurs:

Time	Thread 0	Thread 1
0	`pthread_mutex_lock(&mut0)`	`pthread_mutex_lock(&mut1)`
1	`pthread_mutex_lock(&mut1)`	`pthread_mutex_lock(&mut0)`

What happens?

b. Would this be a problem if the program used busy-waiting (with two flag variables) instead of mutexes?

c. Would this be a problem if the program used semaphores instead of mutexes?

4.9. Some implementations of Pthreads define barriers. The function

```
int pthread_barrier_init(
      pthread_barrier_t*            barrier_p  /* out */,
      const pthread_barrierattr_t*  attr_p     /* in  */,
      unsigned                      count      /* in  */);
```

initializes a barrier object, `barrier_p`. As usual, we'll ignore the second argument and just pass in `NULL`. The last argument indicates the number of threads that must reach the barrier before they can continue. The barrier itself is a call to the function

```
int pthread_barrier_wait(
      pthread_barrier_t*  barrier_p  /* in/out */);
```

As with most other Pthreads objects, there is a destroy function

```
int pthread_barrier_destroy(
      pthread_barrier_t*  barrier_p  /* in/out */);
```

Modify one of the barrier programs from the book's website so that it uses a Pthreads barrier. Find a system with a Pthreads implementation that includes barrier and run your program with various numbers of threads. How does its performance compare to the other implementations?

4.10. Modify one of the programs you wrote in the Programming Assignments that follow so that it uses the scheme outlined in Section 4.8 to time itself. In order to get the time that has elapsed since some point in the past, you can use the macro `GET_TIME` defined in the header file `timer.h` on the book's website.

Note that this will give *wall clock* time, not CPU time. Also note that since it's a macro, it can operate directly on its argument. For example, to implement

```
Store current time in my_start;
```

you would use

```
GET_TIME(my_start);
```

not

```
GET_TIME(&my_start);
```

How will you implement the barrier? How will you implement the following pseudo code?

```
elapsed = Maximum of my_elapsed values;
```

4.11. Give an example of a linked list and a sequence of memory accesses to the linked list in which the following pairs of operations can potentially result in problems:
- **a.** Two deletes executed simultaneously
- **b.** An insert and a delete executed simultaneously
- **c.** A member and a delete executed simultaneously
- **d.** Two inserts executed simultaneously
- **e.** An insert and a member executed simultaneously

4.12. The linked list operations `Insert` and `Delete` consist of two distinct "phases." In the first phase, both operations search the list for either the position of the new node or the position of the node to be deleted. If the outcome of the first phase so indicates, in the second phase a new node is inserted or an existing node is deleted. In fact, it's quite common for linked list programs to split each of these operations into two function calls. For both operations, the first phase involves only read-access to the list; only the second phase modifies the list. Would it be safe to lock the list using a read-lock for the first phase? And then to lock the list using a write-lock for the second phase? Explain your answer.

4.13. Download the various threaded linked list programs from the website. In our examples, we ran a fixed percentage of searches and split the remaining percentage among inserts and deletes.
- **a.** Rerun the experiments with all searches and all inserts.
- **b.** Rerun the experiments with all searches and all deletes.

Is there a difference in the overall run-times? Is insert or delete more expensive?

4.14. Recall that in C a function that takes a two-dimensional array argument must specify the number of columns in the argument list. Thus it is quite common for C programmers to only use one-dimensional arrays, and to write explicit code for converting pairs of subscripts into a single dimension. Modify the

Pthreads matrix-vector multiplication so that it uses a one-dimensional array for the matrix and calls a matrix-vector multiplication function. How does this change affect the run-time?

4.15. Download the source file `pth_mat_vect_rand_split.c` from the book's website. Find a program that does cache profiling (for example, Valgrind [49]) and compile the program according to the instructions in the cache profiler documentation. (with Valgrind you will want a symbol table and full optimization `gcc -g -O2 . . .`). Now run the program according to the instructions in the cache profiler documentation, using input $k \times (k \cdot 10^6)$, $(k \cdot 10^3) \times (k \cdot 10^3)$, and $(k \cdot 10^6) \times k$. Choose k so large that the number of level 2 cache misses is of the order 10^6 for at least one of the input sets of data.

- **a.** How many level 1 cache write-misses occur with each of the three inputs?
- **b.** How many level 2 cache write-misses occur with each of the three inputs?
- **c.** Where do most of the write-misses occur? For which input data does the program have the most write-misses? Can you explain why?
- **d.** How many level 1 cache read-misses occur with each of the three inputs?
- **e.** How many level 2 cache read-misses occur with each of the three inputs?
- **f.** Where do most of the read-misses occur? For which input data does the program have the most read-misses? Can you explain why?
- **g.** Run the program with each of the three inputs, but without using the cache profiler. With which input is the program the fastest? With which input is the program the slowest? Can your observations about cache misses help explain the differences? How?

4.16. Recall the matrix-vector multiplication example with the 8000×8000 input. Suppose that the program is run with four threads, and thread 0 and thread 2 are assigned to different processors. If a cache line contains 64 bytes or eight `doubles`, is it possible for false sharing between threads 0 and 2 to occur for any part of the vector `y`? Why? What about if thread 0 and thread 3 are assigned to different processors–is it possible for false sharing to occur between them for any part of `y`?

4.17. Recall the matrix-vector multiplication example with an $8 \times 8{,}000{,}000$ matrix. Suppose that `doubles` use 8 bytes of memory and that a cache line is 64 bytes. Also suppose that our system consists of two dual-core processors.

- **a.** What is the minimum number of cache lines that are needed to store the vector `y`?
- **b.** What is the maximum number of cache lines that are needed to store the vector `y`?
- **c.** If the boundaries of cache lines always coincide with the boundaries of 8-byte `doubles`, in how many different ways can the components of `y` be assigned to cache lines?
- **d.** If we only consider which pairs of threads share a processor, in how many different ways can four threads be assigned to the processors in our

computer? Here we're assuming that cores on the same processor share a cache.

e. Is there an assignment of components to cache lines and threads to processors that will result in no falses sharing in our example? In other words, is it possible that the threads assigned to one processor will have their components of `y` in one cache line, and the threads assigned to the other processor will have their components in a different cache line?

f. How many assignments of components to cache lines and threads to processors are there?

g. Of these assignments, how many will result in no false sharing?

4.18. **a.** Modify the matrix-vector multiplication program so that it pads the vector `y` when there's a possibility of false sharing. The padding should be done so that if the threads execute in lock-step, there's no possibility that a single cache line containing an element of `y` will be shared by two or more threads. Suppose, for example, that a cache line stores eight `double`s and we run the program with four threads. If we allocate storage for at least 48 `doubles` in `y`, then, on each pass through the `for i` loop, there's no possibility that two threads will simultaneously access the same cache line.

b. Modify the matrix-vector multiplication so that each thread uses private storage for its part of `y` during the `for i` loop. When a thread is done computing its part of `y`, it should copy its private storage into the shared variable.

c. How does the performance of these two alternatives compare to the original program? How do they compare to each other?

4.19. Although `strtok_r` is thread-safe, it has the rather unfortunate property that it gratuitously modifies the input string. Write a tokenizer that is thread-safe and doesn't modify the input string.

4.14 PROGRAMMING ASSIGNMENTS

4.1. Write a Pthreads program that implements the histogram program in Chapter 2.

4.2. Suppose we toss darts randomly at a square dartboard, whose bullseye is at the origin, and whose sides are 2 feet in length. Suppose also that there's a circle inscribed in the square dartboard. The radius of the circle is 1 foot, and it's area is π square feet. If the points that are hit by the darts are uniformly distributed (and we always hit the square), then the number of darts that hit inside the circle should approximately satisfy the equation

$$\frac{\text{number in circle}}{\text{total number of tosses}} = \frac{\pi}{4},$$

since the ratio of the area of the circle to the area of the square is $\pi/4$.

We can use this formula to estimate the value of π with a random number generator:

```
number_in_circle = 0;
for (toss = 0; toss < number_of_tosses; toss++) {
   x = random double between −1 and 1;
   y = random double between −1 and 1;
   distance_squared = x*x + y*y;
   if (distance_squared <= 1) number_in_circle++;
}
pi_estimate = 4*number_in_circle/((double) number_of_tosses);
```

This is called a "Monte Carlo" method, since it uses randomness (the dart tosses).

Write a Pthreads program that uses a Monte Carlo method to estimate π. The main thread should read in the total number of tosses and print the estimate. You may want to use `long long ints` for the number of hits in the circle and the number of tosses, since both may have to be very large to get a reasonable estimate of π.

4.3. Write a Pthreads program that implements the trapezoidal rule. Use a shared variable for the sum of all the threads' computations. Use busy-waiting, mutexes, and semaphores to enforce mutual exclusion in the critical section. What advantages and disadvantages do you see with each approach?

4.4. Write a Pthreads program that finds the average time required by your system to create and terminate a thread. Does the number of threads affect the average time? If so, how?

4.5. Write a Pthreads program that implements a "task queue." The main thread begins by starting a user-specified number of threads that immediately go to sleep in a condition wait. The main thread generates blocks of tasks to be carried out by the other threads; each time it generates a new block of tasks, it awakens a thread with a condition signal. When a thread finishes executing its block of tasks, it should return to a condition wait. When the main thread completes generating tasks, it sets a global variable indicating that there will be no more tasks, and awakens all the threads with a condition broadcast. For the sake of explicitness, make your tasks linked list operations.

4.6. Write a Pthreads program that uses two condition variables and a mutex to implement a read-write lock. Download the online linked list program that uses Pthreads read-write locks, and modify it to use your read-write locks. Now compare the performance of the program when readers are given preference with the program when writers are given preference. Can you make any generalizations?

This page intentionally left blank

CHAPTER

Shared-Memory Programming with OpenMP

5

Like Pthreads, OpenMP is an API for shared-memory parallel programming. The "MP" in OpenMP stands for "multiprocessing," a term that is synonymous with shared-memory parallel computing. Thus, OpenMP is designed for systems in which each thread or process can potentially have access to all available memory, and, when we're programming with OpenMP, we view our system as a collection of cores or CPUs, all of which have access to main memory, as in Figure 5.1.

Although OpenMP and Pthreads are both APIs for shared-memory programming, they have many fundamental differences. Pthreads requires that the programmer explicitly specify the behavior of each thread. OpenMP, on the other hand, sometimes allows the programmer to simply state that a block of code should be executed in parallel, and the precise determination of the tasks and which thread should execute them is left to the compiler and the run-time system. This suggests a further difference between OpenMP and Pthreads, that is, that Pthreads (like MPI) is a library of functions that can be linked to a C program, so any Pthreads program can be used with any C compiler, provided the system has a Pthreads library. OpenMP, on the other hand, requires compiler support for some operations, and hence it's entirely possible that you may run across a C compiler that can't compile OpenMP programs into parallel programs.

These differences also suggest why there are two standard APIs for shared-memory programming: Pthreads is lower level and provides us with the power to program virtually any conceivable thread behavior. This power, however, comes with some associated cost—it's up to us to specify every detail of the behavior of each thread. OpenMP, on the other hand, allows the compiler and run-time system to determine some of the details of thread behavior, so it can be simpler to code some parallel behaviors using OpenMP. The cost is that some low-level thread interactions can be more difficult to program.

OpenMP was developed by a group of programmers and computer scientists who believed that writing large-scale high-performance programs using APIs such as Pthreads was too difficult, and they defined the OpenMP specification so that shared-memory programs could be developed at a higher level. In fact, OpenMP was explicitly designed to allow programmers to *incrementally* parallelize

An Introduction to Parallel Programming. DOI: 10.1016/B978-0-12-374260-5.00005-1

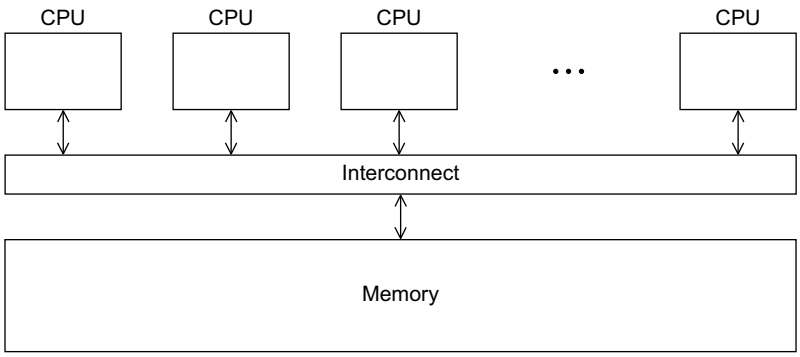


FIGURE 5.1

A shared-memory system

existing serial programs; this is virtually impossible with MPI and fairly difficult with Pthreads.

In this chapter, we'll learn the basics of OpenMP. We'll learn how to write a program that can use OpenMP, and we'll learn how to compile and run OpenMP programs. We'll then learn how to exploit one of the most powerful features of OpenMP: its ability to parallelize many serial `for` loops with only small changes to the source code. We'll then look at some other features of OpenMP: task-parallelism and explicit thread synchronization. We'll also look at some standard problems in shared-memory programming: the effect of cache memories on shared-memory programming and problems that can be encountered when serial code—especially a serial library—is used in a shared-memory program. Let's get started.

5.1 GETTING STARTED

OpenMP provides what's known as a "directives-based" shared-memory API. In C and C++, this means that there are special preprocessor instructions known as `pragma`s. Pragmas are typically added to a system to allow behaviors that aren't part of the basic C specification. Compilers that don't support the `pragma`s are free to ignore them. This allows a program that uses the `pragma`s to run on platforms that don't support them. So, in principle, if you have a carefully written OpenMP program, it can be compiled and run on any system with a C compiler, regardless of whether the compiler supports OpenMP.

Pragmas in C and C++ start with

```
#pragma
```

As usual, we put the pound sign, `#`, in column 1, and like other preprocessor directives, we shift the remainder of the directive so that it is aligned with the rest of the

code. Pragmas (like all preprocessor directives) are, by default, one line in length, so if a pragma won't fit on a single line, the newline needs to be "escaped"—that is, preceded by a backslash \. The details of what follows the #pragma depend entirely on which extensions are being used.

Let's take a look at a *very* simple example, a "hello, world" program that uses OpenMP. See Program 5.1.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <omp.h>
4
5  void Hello(void);  /* Thread function */
6
7  int main(int argc, char* argv[]) {
8     /* Get number of threads from command line */
9     int thread_count = strtol(argv[1], NULL, 10);
10
11 #  pragma omp parallel num_threads(thread_count)
12    Hello();
13
14    return 0;
15 }  /* main */
16
17 void Hello(void) {
18    int my_rank = omp_get_thread_num();
19    int thread_count = omp_get_num_threads();
20
21    printf("Hello from thread %d of %d\n", my_rank, thread_count);
22
23 }  /* Hello */
```

Program 5.1: A "hello,world" program that uses OpenMP

5.1.1 Compiling and running OpenMP programs

To compile this with gcc we need to include the -fopenmp option:[1]

```
$ gcc -g -Wall -fopenmp -o omp_hello omp_hello.c
```

To run the program, we specify the number of threads on the command line. For example, we might run the program with four threads and type

```
$ ./omp_hello 4
```

[1]Some older versions of gcc may not include OpenMP support. Other compilers will, in general, use different command-line options to specify that the source is an OpenMP program. For details on our assumptions about compiler use, see Section 2.9.

If we do this, the output might be

```
Hello from thread 0 of 4
Hello from thread 1 of 4
Hello from thread 2 of 4
Hello from thread 3 of 4
```

However, it should be noted that the threads are competing for access to `stdout`, so there's no guarantee that the output will appear in thread-rank order. For example, the output might also be

```
Hello from thread 1 of 4
Hello from thread 2 of 4
Hello from thread 0 of 4
Hello from thread 3 of 4
```

or

```
Hello from thread 3 of 4
Hello from thread 1 of 4
Hello from thread 2 of 4
Hello from thread 0 of 4
```

or any other permutation of the thread ranks.

If we want to run the program with just one thread, we can type

```
$ ./omp_hello 1
```

and we would get the output

```
Hello from thread 0 of 1
```

5.1.2 The program

Let's take a look at the source code. In addition to a collection of directives, OpenMP consists of a library of functions and macros, so we usually need to include a header file with prototypes and macro definitions. The OpenMP header file is `omp.h`, and we include it in Line 3.

In our Pthreads programs, we specified the number of threads on the command line. We'll also usually do this with our OpenMP programs. In Line 9 we therefore use the `strtol` function from `stdlib.h` to get the number of threads. Recall that the syntax of this function is

```
long strtol(
      const char*  number p     /* in  */,
      char**       end p        /* out */,
      int          base         /* in  */);
```

The first argument is a string—in our example, it's the command-line argument—and the last argument is the numeric base in which the string is represented—in our example, it's base 10. We won't make use of the second argument, so we'll just pass in a `NULL` pointer.

If you've done a little C programming, there's nothing really new up to this point. When we start the program from the command line, the operating system starts a single-threaded process and the process executes the code in the `main` function. However, things get interesting in Line 11. This is our first OpenMP directive, and we're using it to specify that the program should start some threads. Each thread that's forked should execute the `Hello` function, and when the threads return from the call to `Hello`, they should be terminated, and the process should then terminate when it executes the `return` statement.

That's a lot of bang for the buck (or code). If you studied the Pthreads chapter, you'll recall that we had to write a lot of code to fork and join multiple threads: we needed to allocate storage for a special struct for each thread, we used a `for` loop to start each thread, and we used another `for` loop to terminate the threads. Thus, it's immediately evident that OpenMP is higher-level than Pthreads.

We've already seen that `pragma`s in C and C++ start with

```
#  pragma
```

OpenMP `pragma`s always begin with

```
#  pragma omp
```

Our first directive is a `parallel` directive, and, as you might have guessed it specifies that the **structured block** of code that follows should be executed by multiple threads. A structured block is a C statement or a compound C statement with one point of entry and one point of exit, although calls to the function `exit` are allowed. This definition simply prohibits code that branches into or out of the middle of the structured block.

Recollect that **thread** is short for *thread of execution.* The name is meant to suggest a sequence of statements executed by a program. Threads are typically started or **forked** by a process, and they share most of the resources of the process that starts them—for example, access to `stdin` and `stdout`—but each thread has its own stack and program counter. When a thread completes execution it **joins** the process that started it. This terminology comes from diagrams that show threads as directed lines. See Figure 5.2. For more details see Chapters 2 and 4.

At its most basic the `parallel` directive is simply

```
#  pragma omp parallel
```

FIGURE 5.2

A process forking and joining two threads

and the number of threads that run the following structured block of code will be determined by the run-time system. The algorithm used is fairly complicated; see the OpenMP Standard [42] for details. However, if there are no other threads started, the system will typically run one thread on each available core.

As we noted earlier, we'll usually specify the number of threads on the command line, so we'll modify our `parallel` directives with the `num_threads` *clause*. A **clause** in OpenMP is just some text that modifies a directive. The `num_threads` clause can be added to a `parallel` directive. It allows the programmer to specify the number of threads that should execute the following block:

```
#  pragma omp parallel num_threads(thread_count)
```

It should be noted that there may be system-defined limitations on the number of threads that a program can start. The OpenMP Standard doesn't guarantee that this will actually start `thread_count` threads. However, most current systems can start hundreds or even thousands of threads, so unless we're trying to start *a lot* of threads, we will almost always get the desired number of threads.

What actually happens when the program gets to the `parallel` directive? Prior to the `parallel` directive, the program is using a single thread, the process started when the program started execution. When the program reaches the `parallel` directive, the original thread continues executing and `thread_count - 1` additional threads are started. In OpenMP parlance, the collection of threads executing the `parallel` block—the original thread and the new threads—is called a **team**, the original thread is called the **master**, and the additional threads are called **slaves**. Each thread in the team executes the block following the directive, so in our example, each thread calls the `Hello` function.

When the block of code is completed—in our example, when the threads return from the call to `Hello`—there's an **implicit barrier**. This means that a thread that has completed the block of code will wait for all the other threads in the team to complete the block—in our example, a thread that has completed the call to `Hello` will wait for all the other threads in the team to return. When all the threads have completed the block, the slave threads will terminate and the master thread will continue executing the code that follows the block. In our example, the master thread will execute the `return` statement in Line 14, and the program will terminate.

Since each thread has its own stack, a thread executing the `Hello` function will create its own private, local variables in the function. In our example, when the function is called, each thread will get its rank or id and the number of threads in the team by calling the OpenMP functions `omp_get_thread_num` and `omp_get_num_threads`, respectively. The rank or id of a thread is an `int` that is in the range $0, 1, \ldots,$ `thread_count` -1. The syntax for these functions is

```
int omp_get_thread_num(void);
int omp_get_num_threads(void);
```

Since stdout is shared among the threads, each thread can execute the printf statement, printing its rank and the number of threads. As we noted earlier, there is no scheduling of access to stdout, so the actual order in which the threads print their results is nondeterministic.

5.1.3 Error checking

In order to make the code more compact and more readable, our program doesn't do any error checking. Of course, this is dangerous, and, in practice, it's a *very* good idea—one might even say mandatory—to try to anticipate errors and check for them. In this example, we should definitely check for the presence of a command-line argument, and, if there is one, after the call to strtol we should check that the value is positive. We might also check that the number of threads actually created by the parallel directive is the same as thread_count, but in this simple example, this isn't crucial.

A second source of potential problems is the compiler. If the compiler doesn't support OpenMP, it will just ignore the parallel directive. However, the attempt to include omp.h and the calls to omp_get_thread_num and omp_get_num_threads *will* cause errors. To handle these problems, we can check whether the preprocessor macro _OPENMP is defined. If this is defined, we can include omp.h and make the calls to the OpenMP functions. We might make the following modifications to our program.

Instead of simply including omp.h in the line

```
#include <omp.h>
```

we can check for the definition of _OPENMP before trying to include it:

```
#ifdef _OPENMP
#  include <omp.h>
#endif
```

Also, instead of just calling the OpenMP functions, we can first check whether _OPENMP is defined:

```
#  ifdef _OPENMP
      int my_rank = omp_get_thread_num();
      int thread_count = omp_get_num_threads();
#  else
      int my_rank = 0;
      int thread_count = 1;
#  endif
```

Here, if OpenMP isn't available, we assume that the Hello function will be single-threaded. Thus, the single thread's rank will be 0 and the number of threads will be one.

The book's website contains the source for a version of this program that makes these checks. In order to make our code as clear as possible, we'll usually show little, if any, error checking in the code displayed in the text.

5.2 THE TRAPEZOIDAL RULE

Let's take a look at a somewhat more useful (and more complicated) example: the trapezoidal rule for estimating the area under a curve. Recall from Section 3.2 that if $y = f(x)$ is a reasonably nice function, and $a < b$ are real numbers, then we can estimate the area between the graph of $f(x)$, the vertical lines $x = a$ and $x = b$, and the x-axis by dividing the interval $[a,b]$ into n subintervals and approximating the area over each subinterval by the area of a trapezoid. See Figure 5.3 for an example.

Also recall that if each subinterval has the same length and if we define $h = (b - a)/n$, $x_i = a + ih$, $i = 0, 1, \ldots, n$, then our approximation will be

$$h[f(x_0)/2 + f(x_1) + f(x_2) + \cdots + f(x_{n-1}) + f(x_n)/2].$$

Thus, we can implement a serial algorithm using the following code:

```
/* Input:  a, b, n */
h = (b-a)/n;
approx = (f(a) + f(b))/2.0;
for (i = 1; i <= n-1; i++) {
   x_i = a + i*h;
   approx += f(x_i);
}
approx = h*approx;
```

See Section 3.2.1 for details.

5.2.1 A first OpenMP version

Recall that we applied Foster's parallel program design methodology to the trapezoidal rule as described in the following list (see Section 3.2.2).

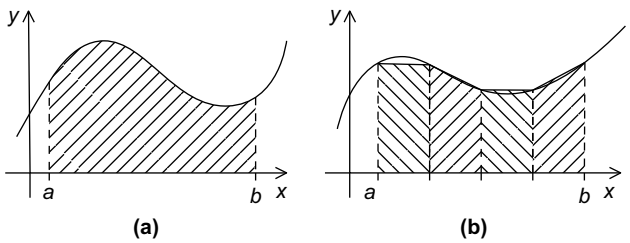


FIGURE 5.3

The trapezoidal rule: (a) area to be estimated and (b) approximate area using trapezoids

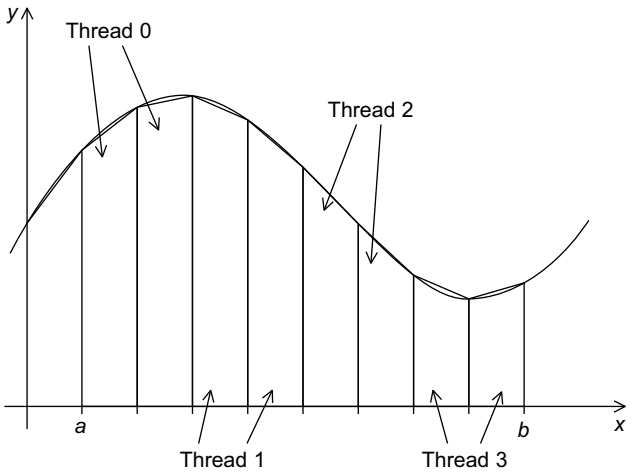


FIGURE 5.4

Assignment of trapezoids to threads

1. We identified two types of tasks:
   a. Computation of the areas of individual trapezoids, and
   b. Adding the areas of trapezoids.
2. There is no communication among the tasks in the first collection, but each task in the first collection communicates with task 1(b).
3. We assumed that there would be many more trapezoids than cores, so we aggregated tasks by assigning a contiguous block of trapezoids to each thread (and a single thread to each core).[2] Effectively, this partitioned the interval $[a,b]$ into larger subintervals, and each thread simply applied the serial trapezoidal rule to its subinterval. See Figure 5.4 for an example.

We aren't quite done, however, since we still need to add up the threads' results. An obvious solution is to use a shared variable for the sum of all the threads' results, and each thread can add its (private) result into the shared variable. We would like to have each thread execute a statement that looks something like

```
global_result += my_result;
```

However, as we've already seen, this can result in an erroneous value for `global_result`—if two (or more) threads attempt to simultaneously execute this statement, the result will be unpredictable. For example, suppose that `global_result` has been initialized to 0, thread 0 has computed `my_result = 1`, and thread 1

[2]Since we were discussing MPI, we actually used *processes* instead of threads.

has computed `my_result = 2`. Furthermore, suppose that the threads execute the statement `global_result += my_result` according to the following timetable:

Time	Thread 0	Thread 1
0	`global_result = 0` to register	finish `my_result`
1	`my_result = 1` to register	`global_result = 0` to register
2	add `my_result` to `global_result`	`my_result = 2` to register
3	store `global_result = 1`	add `my_result` to `global_result`
4		store `global_result = 2`

We see that the value computed by thread 0 (`my_result = 1`) is overwritten by thread 1.

Of course, the actual sequence of events might well be different, but unless one thread finishes the computation `global_result += my_result` before the other starts, the result will be incorrect. Recall that this is an example of a **race condition**: multiple threads are attempting to access a shared resource, at least one of the accesses is an update, and the accesses can result in an error. Also recall that the code that causes the race condition, `global_result += my_result`, is called a **critical section**. A critical section is code executed by multiple threads that updates a shared resource, and the shared resource can only be updated by one thread at a time.

We therefore need some mechanism to make sure that once one thread has started executing `global_result += my_result`, no other thread can start executing this code until the first thread has finished. In Pthreads we used mutexes or semaphores. In OpenMP we can use the `critical` directive

```
#  pragma omp critical
   global_result += my_result;
```

This directive tells the compiler that the system needs to arrange for the threads to have **mutually exclusive** access to the following structured block of code. That is, only one thread can execute the following structured block at a time. The code for this version is shown in Program 5.2. We've omitted any error checking. We've also omitted code for the function $f(x)$.

In the `main` function, prior to Line 16, the code is single-threaded, and it simply gets the number of threads and the input (a, b, and n). In Line 16 the `parallel` directive specifies that the `Trap` function should be executed by `thread_count` threads. After returning from the call to `Trap`, any new threads that were started by the `parallel` directive are terminated, and the program resumes execution with only one thread. The one thread prints the result and terminates.

In the `Trap` function, each thread gets its rank and the total number of threads in the team started by the `parallel` directive. Then each thread determines the following:

1. The length of the bases of the trapezoids (Line 32)
2. The number of trapezoids assigned to each thread (Line 33)

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <omp.h>
4
5  void Trap(double a, double b, int n, double* global_result_p);
6
7  int main(int argc, char* argv[]) {
8     double  global_result = 0.0;
9     double  a, b;
10    int     n;
11    int     thread_count;
12
13    thread_count = strtol(argv[1], NULL, 10);
14    printf("Enter a, b, and n\n");
15    scanf("%lf %lf %d", &a, &b, &n);
16 #  pragma omp parallel num_threads(thread_count)
17    Trap(a, b, n, &global_result);
18
19    printf("With n = %d trapezoids, our estimate\n", n);
20    printf("of the integral from %f to %f = %.14e\n",
21       a, b, global_result);
22    return 0;
23 }  /* main */
24
25 void Trap(double a, double b, int n, double* global_result_p) {
26    double  h, x, my_result;
27    double  local_a, local_b;
28    int  i, local_n;
29    int my_rank = omp_get_thread_num();
30    int thread_count = omp_get_num_threads();
31
32    h = (b-a)/n;
33    local_n = n/thread_count;
34    local_a = a + my_rank*local_n*h;
35    local_b = local_a + local_n*h;
36    my_result = (f(local_a) + f(local_b))/2.0;
37    for (i = 1; i <= local_n-1; i++) {
38      x = local_a + i*h;
39      my_result += f(x);
40    }
41    my_result = my_result*h;
42
43 #  pragma omp critical
44    *global_result_p += my_result;
45 }  /* Trap */
```

Program 5.2: First OpenMP trapezoidal rule program

3. The left and right endpoints of its interval (Lines 34 and 35, respectively)
4. Its contribution to `global_result` (Lines 36–41)

The threads finish by adding in their individual results to `global_result` in Lines 43 and 44.

We use the prefix `local_` for some variables to emphasize that their values may differ from the values of corresponding variables in the `main` function—for example, `local_a` may differ from `a`, although it is the *thread's* left endpoint.

Notice that unless n is evenly divisible by `thread_count`, we'll use fewer than n trapezoids for `global_result`. For example, if $n = 14$ and `thread_count` = 4, each thread will compute

```
local_n = n/thread_count = 14/4 = 3.
```

Thus each thread will only use 3 trapezoids, and `global_result` will be computed with $4 \times 3 = 12$ trapezoids instead of the requested 14. So in the error checking (which isn't shown) we check that n is evenly divisible by `thread_count` by doing something like this:

```
if (n % thread_count != 0) {
   fprintf(stderr, "n must be evenly divisible by thread_count\n");
   exit(0);
}
```

Since each thread is assigned a block of `local_n` trapezoids, the length of each thread's interval will be `local_n*h`, so the left endpoints will be

```
thread 0:  a + 0*local_n*h
thread 1:  a + 1*local_n*h
thread 2:  a + 2*local_h*h
   . . .
```

and in Line 34, we assign

```
local_a = a + my_rank*local_n*h;
```

Furthermore, since the length of each thread's interval will be `local_n*h`, its right endpoint will just be

```
local_b = local_a + local_n*h;
```

5.3 SCOPE OF VARIABLES

In serial programming, the *scope* of a variable consists of those parts of a program in which the variable can be used. For example, a variable declared at the beginning of a C function has "function-wide" scope, that is, it can only be accessed in the body of the function. On the other hand, a variable declared at the beginning of a `.c` file but outside any function has "file-wide" scope, that is, any function in the file

in which the variable is declared can access the variable. In OpenMP, the **scope** of a variable refers to the set of threads that can access the variable in a `parallel` block. A variable that can be accessed by all the threads in the team has **shared** scope, while a variable that can only be accessed by a single thread has **private** scope.

In the "hello, world" program, the variables used by each thread (`my_rank` and `thread_count`) were declared in the `Hello` function, which is called inside the `parallel` block. Consequently, the variables used by each thread are allocated from the thread's (private) stack, and hence all of the variables have private scope. This is *almost* the case in the trapezoidal rule program; since the `parallel` block is just a function call, all of the variables used by each thread in the `Trap` function are allocated from the thread's stack.

However, the variables that are declared in the `main` function (`a`, `b`, `n`, `global_result`, and `thread_count`) are all accessible to all the threads in the team started by the `parallel` directive. Hence, the *default* scope for variables declared before a `parallel` block is shared. In fact, we've made implicit use of this: each thread in the team gets the values of `a`, `b`, and `n` from the call to `Trap`. Since this call takes place in the `parallel` block, it's essential that each thread has access to `a`,`b`, and `n` when their values are copied into the corresponding formal arguments.

Furthermore, in the `Trap` function, although `global_result_p` is a private variable, it refers to the variable `global_result` which was declared in `main` before the `parallel` directive, and the value of `global_result` is used to store the result that's printed out after the `parallel` block. So in the code

```
*global_result_p += my_result;
```

it's essential that `*global_result_p` have shared scope. If it were private to each thread, there would be no need for the `critical` directive. Furthermore, if it were private, we would have a hard time determining the value of `global_result` in `main` after completion of the `parallel` block.

To summarize, then, variables that have been declared before a `parallel` directive have shared scope among the threads in the team, while variables declared in the block (e.g., local variables in functions) have private scope. Furthermore, the value of a shared variable at the beginning of the `parallel` block is the same as the value before the block, and, after completion of the `parallel` block, the value of the variable is the value at the end of the block.

We'll shortly see that the *default* scope of a variable can change with other directives, and that OpenMP provides clauses to modify the default scope.

5.4 THE REDUCTION CLAUSE

If we developed a serial implementation of the trapezoidal rule, we'd probably use a slightly different function prototype. Rather than

```
void Trap(double a, double b, int n, double* global_result_p);
```

we would probably define

```
double Trap(double a, double b, int n);
```

and our function call would be

```
global_result = Trap(a, b, n);
```

This is somewhat easier to understand and probably more attractive to all but the most fanatical believers in pointers.

We resorted to the pointer version because we needed to add each thread's local calculation to get `global_result`. However, we might prefer the following function prototype:

```
double Local_trap(double a, double b, int n);
```

With this prototype, the body of `Local_trap` would be the same as the `Trap` function in Program 5.2, except that there would be no critical section. Rather, each thread would return its part of the calculation, the final value of its `my_result` variable. If we made this change, we might try modifying our `parallel` block so that it looks like this:

```
   global_result = 0.0;
#  pragma omp parallel num_threads(thread_count)
   {
#     pragma omp critical
      global_result += Local_trap(double a, double b, int n);
   }
```

Can you see a problem with this code? It should give the correct result. However, since we've specified that the critical section is

```
global_result += Local_trap(double a, double b, int n);
```

the call to `Local_trap` can only be executed by one thread at a time, and, effectively, we're forcing the threads to execute the trapezoidal rule sequentially. If we check the run-time of this version, it may actually be *slower* with multiple threads than one thread (see Exercise 5.3).

We can avoid this problem by declaring a private variable inside the `parallel` block and moving the critical section after the function call:

```
   global_result = 0.0;
#  pragma omp parallel num_threads(thread_count)
   {
      double my_result = 0.0;  /* private */
      my_result += Local_trap(double a, double b, int n);
#     pragma omp critical
      global_result += my_result;
   }
```

Now the call to `Local_trap` is outside the critical section, and the threads can execute their calls simultaneously. Furthermore, since `my_result` is declared in the

parallel block, it's private, and before the critical section each thread will store its part of the calculation in its my_result variable.

OpenMP provides a cleaner alternative that also avoids serializing execution of Local_trap: we can specify that global_result is a *reduction* variable. A **reduction operator** is a binary operation (such as addition or multiplication) and a **reduction** is a computation that repeatedly applies the same reduction operator to a sequence of operands in order to get a single result. Furthermore, all of the intermediate results of the operation should be stored in the same variable: the **reduction variable**. For example, if A is an array of n ints, the computation

```
int sum = 0;
for (i = 0; i < n; i++)
   sum += A[i];
```

is a reduction in which the reduction operator is addition.

In OpenMP it may be possible to specify that the result of a reduction is a reduction variable. To do this, a reduction clause can be added to a parallel directive. In our example, we can modify the code as follows:

```
   global_result = 0.0;
#  pragma omp parallel num_threads(thread_count) \
      reduction(+: global_result)
   global_result += Local_trap(double a, double b, int n);
```

First note that the parallel directive is two lines long. Recall that C preprocessor directives are, by default, only one line long, so we need to "escape" the newline character by putting a backslash (\) immediately before it.

The code specifies that global_result is a reduction variable and the plus sign ("+") indicates that the reduction operator is addition. Effectively, OpenMP creates a private variable for each thread, and the run-time system stores each thread's result in this private variable. OpenMP also creates a critical section and the values stored in the private variables are added in this critical section. Thus, the calls to Local_trap can take place in parallel.

The syntax of the reduction clause is

```
reduction(<operator>: <variable list>)
```

In C, operator can be any one of the operators +, *, −, &, |, ^, &&, ||, although the use of subtraction is a bit problematic, since subtraction isn't associative or commutative. For example, the serial code

```
result = 0;
for (i = 1; i <= 4; i++)
   result -= i;
```

stores the value -10 in result. If, however, we split the iterations among two threads, with thread 0 subtracting 1 and 2 and thread 1 subtracting 3 and 4, then thread 0 will compute -3 and thread 1 will compute -7 and, of course, $-3-(-7)=4$.

In principle, the compiler should determine that the threads' individual results should actually be added ($-3 + (-7) = -10$), and, in practice, this seems to be the case. However, the OpenMP Standard [42] doesn't seem to guarantee this.

It should also be noted that if a reduction variable is a `float` or a `double`, the results may differ slightly when different numbers of threads are used. This is due to the fact that floating point arithmetic isn't associative. For example, if a, b, and c are **float**s, then $(a + b) + c$ may not be exactly equal to $a + (b + c)$. See Exercise 5.5.

When a variable is included in a `reduction` clause, the variable itself is shared. However, a private variable is created for each thread in the team. In the `parallel` block each time a thread executes a statement involving the variable, it uses the private variable. When the `parallel` block ends, the values in the private variables are combined into the shared variable. Thus, our latest version of the code

```
   global_result = 0.0;
#  pragma omp parallel num_threads(thread_count) \
      reduction(+: global_result)
   global_result += Local_trap(double a, double b, int n);
```

effectively executes code that is identical to our previous version:

```
   global_result = 0.0;
#  pragma omp parallel num_threads(thread_count)
   {
      double my_result = 0.0;  /* private */
      my_result += Local_trap(double a, double b, int n);
#     pragma omp critical
      global_result += my_result;
   }
```

One final point to note is that the threads' private variables are initialized to 0. This is analogous to our initializing `my_result` to zero. In general, the private variables created for a `reduction` clause are initialized to the *identity value* for the operator. For example, if the operator is multiplication, the private variables would be initialized to 1.

5.5 THE `parallel for` DIRECTIVE

As an alternative to our explicit parallelization of the trapezoidal rule, OpenMP provides the `parallel for` directive. Using it, we can parallelize the serial trapezoidal rule

```
h = (b-a)/n;
approx = (f(a) + f(b))/2.0;
for (i = 1; i <= n-1; i++)
   approx += f(a + i*h);
approx = h*approx;
```

by simply placing a directive immediately before the for loop:

```
   h = (b-a)/n;
   approx = (f(a) + f(b))/2.0;
#  pragma omp parallel for num_threads(thread_count) \
      reduction(+: approx)
   for (i = 1; i <= n-1; i++)
      approx += f(a + i*h);
   approx = h*approx;
```

Like the parallel directive, the parallel for directive forks a team of threads to execute the following structured block. However, the structured block following the parallel for directive must be a for loop. Furthermore, with the parallel for directive the system parallelizes the for loop by dividing the iterations of the loop among the threads. The parallel for directive is therefore very different from the parallel directive, because in a block that is preceded by a parallel directive, in general, the work must be divided among the threads by the threads themselves.

In a for loop that has been parallelized with a parallel for directive, the default partitioning, that is, of the iterations among the threads is up to the system. However, most systems use roughly a block partitioning, that is, if there are m iterations, then roughly the first $m/$thread_count are assigned to thread 0, the next $m/$thread_count are assigned to thread 1, and so on.

Note that it was essential that we made approx a reduction variable. If we hadn't, it would have been an ordinary shared variable, and the body of the loop

```
approx += f(a + i*h);
```

would be an unprotected critical section.

However, speaking of scope, the default scope for all variables in a parallel directive is shared, but in our parallel for if the loop variable i were shared, the variable update, i++, would also be an unprotected critical section. Hence, in a loop that is parallelized with a parallel for directive, the default scope of the loop variable is *private*; in our code, each thread in the team has its own copy of i.

5.5.1 Caveats

This is truly wonderful: It may be possible to parallelize a serial program that consists of one large for loop by just adding a single parallel for directive. It may be possible to incrementally parallelize a serial program that has many for loops by successively placing parallel for directives before each loop.

However, things may not be quite as rosy as they seem. There are several caveats associated with the use of the parallel for directive. First, OpenMP will only parallelize for loops. It won't parallelize while loops or do-while loops. This may not seem to be too much of a limitation, since any code that uses a while loop or a do-while loop can be converted to equivalent code that uses a for loop instead.

However, OpenMP will only parallelize `for` loops for which the number of iterations can be determined

- from the `for` statement itself (that is, the code `for (. . . ; . . . ; . . .)`), and
- prior to execution of the loop.

For example, the "infinite loop"

```
for ( ; ; ) {
   . . .
}
```

cannot be parallelized. Similarly, the loop

```
for (i = 0; i < n; i++) {
   if ( . . . ) break;
   . . .
}
```

cannot be parallelized, since the number of iterations can't be determined from the `for` statement alone. This `for` loop is also not a structured block, since the `break` adds another point of exit from the loop.

In fact, OpenMP will only parallelize `for` loops that are in **canonical form**. Loops in canonical form take one of the forms shown in Program 5.3. The variables and expressions in this template are subject to some fairly obvious restrictions:

- The variable `index` must have integer or pointer type (e.g., it can't be a **float**).
- The expressions `start`, `end`, and `incr` must have a compatible type. For example, if `index` is a pointer, then `incr` must have integer type.
- The expressions `start`, `end`, and `incr` must not change during execution of the loop.
- During execution of the loop, the variable `index` can only be modified by the "increment expression" in the `for` statement.

```
                                          index++
                                          ++index
                          index < end     index--
                          index <= end    --index
for | index = start   ;   index >= end  ; index += incr
                          index > end     index -= incr
                                          index = index + incr
                                          index = incr + index
                                          index = index - incr
```

Program 5.3: Legal forms for parallelizable `for` statements

These restrictions allow the run-time system to determine the number of iterations prior to execution of the loop.

The sole exception to the rule that the run-time system must be able to determine the number of iterations prior to execution is that there *can* be a call to exit in the body of the loop.

5.5.2 Data dependences

If a for loop fails to satisfy one of the rules outlined in the preceding section, the compiler will simply reject it. For example, suppose we try to compile a program with the following linear search function:

```
1     int Linear_search(int key, int A[], int n) {
2        int i;
3        /* thread_count is global */
4     #  pragma omp parallel for num_threads(thread_count)
5        for (i = 0; i < n; i++)
6           if (A[i] == key) return i;
7        return -1;  /* key not in list */
8     }
```

The gcc compiler reports:

```
Line 6: error: invalid exit from OpenMP structured block
```

A more insidious problem occurs in loops in which the computation in one iteration depends on the results of one or more previous iterations. As an example, consider the following code, which computes the first *n* fibonacci numbers:

```
fibo[0] = fibo[1] = 1;
for (i = 2; i < n; i++)
   fibo[i] = fibo[i-1] + fibo[i-2];
```

Although we may be suspicious that something isn't quite right, let's try parallellizing the for loop with a parallel for directive:

```
   fibo[0] = fibo[1] = 1;
#  pragma omp parallel for num_threads(thread_count)
   for (i = 2; i < n; i++)
      fibo[i] = fibo[i-1] + fibo[i-2];
```

The compiler will create an executable without complaint. However, if we try running it with more than one thread, we may find that the results are, at best, unpredictable. For example, on one of our systems if we try using two threads to compute the first 10 Fibonacci numbers, we sometimes get

$$1\ 1\ 2\ 3\ 5\ 8\ 13\ 21\ 34\ 55,$$

which is correct. However, we also occasionally get

$$1\ 1\ 2\ 3\ 5\ 8\ 0\ 0\ 0\ 0.$$

What happened? It appears that the run-time system assigned the computation of `fibo[2]`, `fibo[3]`, `fibo[4]`, and `fibo[5]` to one thread, while `fibo[6]`, `fibo[7]`, `fibo[8]`, and `fibo[9]` were assigned to the other. (Remember the loop starts with `i = 2`.) In some runs of the program, everything is fine because the thread that was assigned `fibo[2]`, `fibo[3]`, `fibo[4]`, and `fibo[5]` finishes its computations before the other thread starts. However, in other runs, the first thread has evidently not computed `fibo[4]` and `fibo[5]` when the second computes `fibo[6]`. It appears that the system has initialized the entries in `fibo` to 0, and the second thread is using the values `fibo[4] = 0` and `fibo[5] = 0` to compute `fibo[6]`. It then goes on to use `fibo[5] = 0` and `fibo[6] = 0` to compute `fibo[7]`, and so on.

We see two important points here:

1. OpenMP compilers don't check for dependences among iterations in a loop that's being parallelized with a `parallel for` directive. It's up to us, the programmers, to identify these dependences.
2. A loop in which the results of one or more iterations depend on other iterations *cannot,* in general, be correctly parallelized by OpenMP.

The dependence of the computation of `fibo[6]` on the computation of `fibo[5]` is called a **data dependence**. Since the value of `fibo[5]` is calculated in one iteration, and the result is used in a subsequent iteration, the dependence is sometimes called a **loop-carried dependence**.

5.5.3 Finding loop-carried dependences

Perhaps the first thing to observe is that when we're attempting to use a `parallel for` directive, we only need to worry about loop-carried dependences. We don't need to worry about more general data dependences. For example, in the loop

```
1     for (i = 0; i < n; i++) {
2        x[i] = a + i*h;
3        y[i] = exp(x[i]);
4     }
```

there is a data dependence between Lines 2 and 3. However, there is no problem with the parallelization

```
1  #  pragma omp parallel for num_threads(thread_count)
2     for (i = 0; i < n; i++) {
3        x[i] = a + i*h;
4        y[i] = exp(x[i]);
5     }
```

since the computation of `x[i]` and its subsequent use will always be assigned to the same thread.

Also observe that at least one of the statements must write or update the variable in order for the statements to represent a dependence, so in order to detect a loop-carried dependence, we should only concern ourselves with variables that are updated

by the loop body. That is, we should look for variables that are read or written in one iteration, and written in another. Let's look at a couple of examples.

5.5.4 Estimating π

One way to get a numerical approximation to π is to use many terms in the formula[3]

$$\pi = 4\left[1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots\right] = 4\sum_{k=0}^{\infty}\frac{(-1)^k}{2k+1}.$$

We can implement this formula in serial code with

```
1         double factor = 1.0;
2         double sum = 0.0;
3         for (k = 0; k < n; k++) {
4            sum += factor/(2*k+1);
5            factor = -factor;
6         }
7         pi_approx = 4.0*sum;
```

(Why is it important that `factor` is a `double` instead of an `int` or a `long`?)

How can we parallelize this with OpenMP? We might at first be inclined to do something like this:

```
1         double factor = 1.0;
2         double sum = 0.0;
3   #     pragma omp parallel for num_threads(thread_count) \
4            reduction(+:sum)
5         for (k = 0; k < n; k++) {
6            sum += factor/(2*k+1);
7            factor = -factor;
8         }
9         pi_approx = 4.0*sum;
```

However, it's pretty clear that the update to `factor` in Line 7 in iteration `k` and the subsequent increment of `sum` in Line 6 in iteration `k+1` is an instance of a loop-carried dependence. If iteration `k` is assigned to one thread and iteration `k+1` is assigned to another thread, there's no guarantee that the value of `factor` in Line 6 will be correct. In this case we can fix the problem by examining the series

$$\sum_{k=0}^{\infty}\frac{(-1)^k}{2k+1}.$$

[3]This is by no means the best method for approximating π, since it requires a *lot* of terms to get a reasonably accurate result. However, we're more interested in the formula itself than the actual estimate.

We see that in iteration k the value of `factor` should be $(-1)^k$, which is $+1$ if k is even and -1 if k is odd, so if we replace the code

```
1           sum += factor/(2*k+1);
2           factor = -factor;
```

by

```
1           if (k % 2 == 0)
2              factor = 1.0;
3           else
4              factor = -1.0;
5           sum += factor/(2*k+1);
```

or, if you prefer the ?: operator,

```
1           factor = (k % 2 == 0) ? 1.0 : -1.0;
2           sum += factor/(2*k+1);
```

we will eliminate the loop dependency.

However, things still aren't quite right. If we run the program on one of our systems with just two threads and $n = 1000$, the result is consistently wrong. For example,

```
1     With n = 1000 terms and 2 threads,
2        Our estimate of pi = 2.97063289263385
3     With n = 1000 terms and 2 threads,
4        Our estimate of pi = 3.22392164798593
```

On the other hand, if we run the program with only one thread, we always get

```
1     With n = 1000 terms and 1 threads,
2        Our estimate of pi = 3.14059265383979
```

What's wrong here?

Recall that in a block that has been parallelized by a `parallel for` directive, by default any variable declared before the loop—with the sole exception of the loop variable—is shared among the threads. So `factor` is shared and, for example, thread 0 might assign it the value 1, but before it can use this value in the update to `sum`, thread 1 could assign it the value -1. Therefore, in addition to eliminating the loop-carried dependence in the calculation of `factor`, we need to insure that each thread has its own copy of `factor`. That is, in order to make our code correct, we need to also insure that `factor` has private scope. We can do this by adding a `private` clause to the `parallel for` directive.

```
1        double sum = 0.0;
2  #     pragma omp parallel for num_threads(thread_count) \
3           reduction(+:sum) private(factor)
4        for (k = 0; k < n; k++) {
5           if (k % 2 == 0)
```

```
6              factor = 1.0;
7           else
8              factor = -1.0;
9           sum += factor/(2*k+1);
10       }
```

The `private` clause specifies that for each variable listed inside the parentheses, a private copy is to be created for each thread. Thus, in our example, each of the `thread_count` threads will have its own copy of the variable `factor`, and hence the updates of one thread to `factor` won't affect the value of `factor` in another thread.

It's important to remember that the value of a variable with private scope is unspecified at the beginning of a `parallel` block or a `parallel for` block. Its value is also unspecified after completion of a `parallel` or `parallel for` block. So, for example, the output of the first `printf` statement in the following code is unspecified, since it prints the private variable `x` before it's explicitly initialized. Similarly, the output of the final `printf` is unspecified, since it prints `x` after the completion of the `parallel` block.

```
1     int x = 5;
2  #  pragma omp parallel num_threads(thread_count) \
3        private(x)
4     {
5        int my_rank = omp_get_thread_num();
6        printf("Thread %d > before initialization, x = %d\n",
7              my_rank, x);
8        x = 2*my_rank + 2;
9        printf("Thread %d > after initialization, x = %d\n",
10             my_rank, x);
11    }
12    printf("After parallel block, x = %d\n", x);
```

5.5.5 More on scope

Our problem with the variable `factor` is a common one. We usually need to think about the scope of each variable in a `parallel` block or a `parallel for` block. Therefore, rather than letting OpenMP decide on the scope of each variable, it's a very good practice for us as programmers to specify the scope of each variable in a block. In fact, OpenMP provides a clause that will explicitly require us to do this: the `default` clause. If we add the clause

```
default(none)
```

to our `parallel` or `parallel for` directive, then the compiler will require that we specify the scope of each variable we use in the block and that has been declared outside the block. (Variables that are declared within the block are always private, since they are allocated on the thread's stack.)

For example, using a `default(none)` clause, our calculation of π could be written as follows:

```
      double sum = 0.0;
#     pragma omp parallel for num_threads(thread_count) \
         default(none) reduction(+:sum) private(k, factor) \
         shared(n)
      for (k = 0; k < n; k++) {
         if (k % 2 == 0)
            factor = 1.0;
         else
            factor = -1.0;
         sum += factor/(2*k+1);
      }
```

In this example, we use four variables in the `for` loop. With the default clause, we need to specify the scope of each. As we've already noted, `sum` is a reduction variable (which has properties of both private and shared scope). We've also already noted that `factor` and the loop variable `k` should have private scope. Variables that are never updated in the `parallel` or `parallel for` block, such as `n` in this example, can be safely shared. Recall that unlike `private` variables, `shared` variables have the same value in the `parallel` or `parallel for` block that they had before the block, and their value after the block is the same as their last value in the block. Thus, if `n` were initialized before the block to 1000, it would retain this value in the `parallel for` statement, and since the value isn't changed in the `for` loop, it would retain this value after the loop has completed.

5.6 MORE ABOUT LOOPS IN OPENMP: SORTING

5.6.1 Bubble sort

Recollect that the serial *bubble sort* algorithm for sorting a list of integers can be implemented as follows:

```
for (list_length = n; list_length >= 2; list_length--)
   for (i = 0; i < list_length-1; i++)
      if (a[i] > a[i+1]) {
         tmp = a[i];
         a[i] = a[i+1];
         a[i+1] = tmp;
      }
```

Here, `a` stores n `int`s and the algorithm sorts them in increasing order. The outer loop first finds the largest element in the list and stores it in `a[n-1]`; it then finds the next-to-the-largest element and stores it in `a[n-2]`, and so on. So, effectively, the first pass is working with the full n-element list. The second is working with all of the elements, except the largest; it's working with an $n-1$-element list, and so on.

The inner loop compares consecutive pairs of elements in the current list. When a pair is out of order (`a[i] > a[i+1]`) it swaps them. This process of swapping will move the largest element to the last slot in the "current" list, that is, the list consisting of the elements

```
a[0], a[1], . . . , a[list_length–1]
```

It's pretty clear that there's a loop-carried dependence in the outer loop; in any iteration of the outer loop the contents of the current list depends on the previous iterations of the outer loop. For example, if at the start of the algorithm `a = 3, 4, 1, 2`, then the second iteration of the outer loop should work with the list `3, 1, 2`, since the 4 should be moved to the last position by the first iteration. But if the first two iterations are executing simultaneously, it's possible that the effective list for the second iteration will contain 4.

The loop-carried dependence in the inner loop is also fairly easy to see. In iteration i the elements that are compared depend on the outcome of iteration $i-1$. If in iteration $i-1$, `a[i–1]` and `a[i]` are not swapped, then iteration i should compare `a[i]` and `a[i+1]`. If, on the other hand, iteration $i-1$ swaps `a[i–1]` and `a[i]`, then iteration i should be comparing the original `a[i–1]` (which is now `a[i]`) and `a[i+1]`. For example, suppose the current list is {3,1,2}. Then when $i = 1$, we should compare 3 and 2, but if the $i = 0$ and the $i = 1$ iterations are happening simultaneously, it's entirely possible that the $i = 1$ iteration will compare 1 and 2.

It's also not at all clear how we might remove either loop-carried dependence without completely rewriting the algorithm. It's important to keep in mind that even though we can always find loop-carried dependences, it may be difficult or impossible to remove them. The `parallel for` directive is not a universal solution to the problem of parallelizing `for` loops.

5.6.2 Odd-even transposition sort

Odd-even transposition sort is a sorting algorithm that's similar to bubble sort, but that has considerably more opportunities for parallelism. Recall from Section 3.7.1 that serial odd-even transposition sort can be implemented as follows:

```
for (phase = 0; phase < n; phase++)
   if (phase % 2 == 0)
      for (i = 1; i < n; i += 2)
         if (a[i–1] > a[i]) Swap(&a[i–1],&a[i]);
   else
      for (i = 1; i < n–1; i += 2)
         if (a[i] > a[i+1]) Swap(&a[i], &a[i+1]);
```

The list `a` stores n `ints`, and the algorithm sorts them into increasing order. During an "even phase" (`phase % 2 == 0`), each odd-subscripted element, `a[i]`, is compared to the element to its "left," `a[i–1]`, and if they're out of order, they're swapped. During an "odd" phase, each odd-subscripted element is compared to the element to its right, and if they're out of order, they're swapped. A theorem guarantees that after n phases, the list will be sorted.

Table 5.1 Serial Odd-Even Transposition Sort

	Subscript in Array						
Phase	**0**		**1**		**2**		**3**
0	9	↔	7		8	↔	6
	7		9		6		8
1	7		9	↔	6		8
	7		6		9		8
2	7	↔	6		9	↔	8
	6		7		8		9
3	6		7	↔	8		9
	6		7		8		9

As a brief example, suppose `a = {9, 7, 8, 6}`. Then the phases are shown in Table 5.1. In this case, the final phase wasn't necessary, but the algorithm doesn't bother checking whether the list is already sorted before carrying out each phase.

It's not hard to see that the outer loop has a loop-carried dependence. As an example, suppose as before that `a = {9, 7, 8, 6}`. Then in `phase 0` the inner loop will compare elements in the pairs $(9,7)$ and $(8,6)$, and both pairs are swapped. So for `phase 1` the list should be `{7, 9, 6, 8}`, and during phase 1 the elements in the pair $(9,6)$ should be compared and swapped. However, if `phase 0` and `phase 1` are executed simultaneously, the pair that's checked in `phase 1` might be $(7,8)$, which is in order. Furthermore, it's not clear how one might eliminate this loop-carried dependence, so it would appear that parallelizing the outer `for` loop isn't an option.

The *inner* `for` loops, however, don't appear to have any loop-carried dependences. For example, in an even phase loop, variable i will be odd, so for two distinct values of i, say $i = j$ and $i = k$, the pairs $\{j-1,j\}$ and $\{k-1,k\}$ will be be disjoint. The comparison and possible swaps of the pairs (`a[j-1]`, `a[j]`) and (`a[k-1]`, `a[k]`) can therefore proceed simultaneously.

Thus, we could try to parallelize odd-even transposition sort using the code shown in Program 5.4, but there are a couple of potential problems. First, although any iteration of, say, one even phase doesn't depend on any other iteration of that phase, we've already noted that this is not the case for iterations in `phase` p and `phase` $p+1$. We need to be sure that all the threads have finished `phase` p before any thread starts `phase` $p+1$. However, like the `parallel` directive, the `parallel for` directive has an implicit barrier at the end of the loop, so none of the threads will proceed to the next phase, `phase` $p+1$, until all of the threads have completed the current phase, `phase` p.

A second potential problem is the overhead associated with forking and joining the threads. The OpenMP implementation *may* fork and join `thread_count` threads on *each* pass through the body of the outer loop. The first row of Table 5.2 shows

```
1      for (phase = 0; phase < n; phase++) {
2         if (phase % 2 == 0)
3  #         pragma omp parallel for num_threads(thread_count) \
4               default(none) shared(a, n) private(i, tmp)
5            for (i = 1; i < n; i += 2) {
6               if (a[i-1] > a[i]) {
7                  tmp = a[i-1];
8                  a[i-1] = a[i];
9                  a[i] = tmp;
10              }
11           }
12        else
13 #         pragma omp parallel for num_threads(thread_count) \
14             default(none) shared(a, n) private(i, tmp)
15           for (i = 1; i < n-1; i += 2) {
16              if (a[i] > a[i+1]) {
17                 tmp = a[i+1];
18                 a[i+1] = a[i];
19                 a[i] = tmp;
20              }
21           }
22     }
```

Program 5.4: First OpenMP implementation of odd-even sort

Table 5.2 Odd-Even Sort with Two `parallel for` Directives and Two `for` Directives (times are in seconds)

thread_count	1	2	3	4
Two `parallel for` directives	0.770	0.453	0.358	0.305
Two `for` directives	0.732	0.376	0.294	0.239

run-times for 1, 2, 3, and 4 threads on one of our systems when the input list contained 20,000 elements.

These aren't terrible times, but let's see if we can do better. Each time we execute one of the inner loops, we use the same number of threads, so it would seem to be superior to fork the threads once and reuse the same team of threads for each execution of the inner loops. Not surprisingly, OpenMP provides directives that allow us to do just this. We can fork our team of `thread_count` threads *before* the outer loop with a `parallel` directive. Then, rather than forking a new team of threads with each execution of one of the inner loops, we use a `for` directive, which tells OpenMP to parallelize the `for` loop with the existing team of threads. This modification to the original OpenMP implementation is shown in Program 5.5

The `for` directive, unlike the `parallel for` directive, doesn't fork any threads. It uses whatever threads have already been forked in the enclosing `parallel` block.

```
1  #  pragma omp parallel num_threads(thread_count) \
2        default(none) shared(a, n) private(i, tmp, phase)
3     for (phase = 0; phase < n; phase++) {
4        if (phase % 2 == 0)
5  #        pragma omp for
6           for (i = 1; i < n; i += 2) {
7              if (a[i-1] > a[i]) {
8                 tmp = a[i-1];
9                 a[i-1] = a[i];
10                a[i] = tmp;
11             }
12          }
13       else
14 #        pragma omp for
15          for (i = 1; i < n-1; i += 2) {
16             if (a[i] > a[i+1]) {
17                tmp = a[i+1];
18                a[i+1] = a[i];
19                a[i] = tmp;
20             }
21          }
22    }
```

Program 5.5: Second OpenMP implementation of odd-even sort

There *is* an implicit barrier at the end of the loop. The results of the code—the final list—will therefore be the same as the results obtained from the original parallelized code.

Run-times for this second version of odd-even sort are in the second row of Table 5.2. When we're using two or more threads, the version that uses two `for` directives is at least 17% faster than the version that uses two `parallel for` directives, so for this system the slight effort involved in making the change is well worth it.

5.7 SCHEDULING LOOPS

When we first encountered the `parallel for` directive, we saw that the exact assignment of loop iterations to threads is system dependent. However, most OpenMP implementations use roughly a block partitioning: if there are n iterations in the serial loop, then in the parallel loop the first n/`thread_count` are assigned to thread 0, the next n/`thread_count` are assigned to thread 1, and so on. It's not difficult to think of situations in which this assignment of iterations to threads would be less than optimal. For example, suppose we want to parallelize the loop

```
sum = 0.0;
for (i = 0; i <= n; i++)
   sum += f(i);
```

Also suppose that the time required by the call to f is proportional to the size of the argument `i`. Then a block partitioning of the iterations will assign much more work

to thread `thread_count` − 1 than it will assign to thread 0. A better assignment of work to threads might be obtained with a **cyclic** partitioning of the iterations among the threads. In a cyclic partitioning, the iterations are assigned, one at a time, in a "round-robin" fashion to the threads. Suppose $t =$ `thread_count`. Then a cyclic partitioning will assign the iterations as follows:

Thread	Iterations
0	0, n/t, 2n/t, ...
1	1, n/t + 1, 2n/t + 1, ...
⋮	⋮
t − 1	t − 1, n/t + t − 1, 2n/t + t − 1, ...

To get a feel for how drastically this can affect performance, we wrote a program in which we defined

```
double f(int i) {
   int j, start = i*(i+1)/2, finish = start + i;
   double return_val = 0.0;

   for (j = start; j <= finish; j++) {
      return_val += sin(j);
   }
   return return_val;
}  /* f */
```

The call $f(i)$ calls the sine function i times, and, for example, the time to execute $f(2i)$ requires approximately twice as much time as the time to execute $f(i)$.

When we ran the program with $n = 10{,}000$ and one thread, the run-time was 3.67 seconds. When we ran the program with two threads and the default assignment—iterations 0–5000 on thread 0 and iterations 5001–10,000 on thread 1—the run-time was 2.76 seconds. This is a speedup of only 1.33. However, when we ran the program with two threads and a cyclic assignment, the run-time was decreased to 1.84 seconds. This is a speedup of 1.99 over the one-thread run and a speedup of 1.5 over the two-thread block partition!

We can see that a good assignment of iterations to threads can have a very significant effect on performance. In OpenMP, assigning iterations to threads is called **scheduling**, and the `schedule` clause can be used to assign iterations in either a `parallel for` or a `for` directive.

5.7.1 The `schedule` clause

In our example, we already know how to obtain the default schedule: we just add a `parallel for` directive with a `reduction` clause:

```
      sum = 0.0;
#     pragma omp parallel for num_threads(thread_count) \
         reduction(+:sum)
```

```
      for (i = 0; i <= n; i++)
         sum += f(i);
```

To get a cyclic schedule, we can add a `schedule` clause to the `parallel for` directive:

```
      sum = 0.0;
#     pragma omp parallel for num_threads(thread_count) \
         reduction(+:sum) schedule(static,1)
      for (i = 0; i <= n; i++)
         sum += f(i);
```

In general, the `schedule` clause has the form

```
schedule(<type> [, <chunksize>])
```

The `type` can be any one of the following:

- `static`. The iterations can be assigned to the threads before the loop is executed.
- `dynamic` or `guided`. The iterations are assigned to the threads while the loop is executing, so after a thread completes its current set of iterations, it can request more from the run-time system.
- `auto`. The compiler and/or the run-time system determine the schedule.
- `runtime`. The schedule is determined at run-time.

The `chunksize` is a positive integer. In OpenMP parlance, a **chunk** of iterations is a block of iterations that would be executed consecutively in the serial loop. The number of iterations in the block is the `chunksize`. Only `static`, `dynamic`, and `guided` schedules can have a `chunksize`. This determines the details of the schedule, but its exact interpretation depends on the `type`.

5.7.2 The `static` schedule type

For a `static` schedule, the system assigns chunks of `chunksize` iterations to each thread in a round-robin fashion. As an example, suppose we have 12 iterations, $0, 1, \ldots, 11$, and three threads. Then if `schedule(static,1)` is used in the `parallel for` or `for` directive, we've already seen that the iterations will be assigned as

Thread 0: $0, 3, 6, 9$

Thread 1: $1, 4, 7, 10$

Thread 2: $2, 5, 8, 11$

If `schedule(static,2)` is used, then the iterations will be assigned as

Thread 0: $0, 1, 6, 7$

Thread 1: $2, 3, 8, 9$

Thread 2: $4, 5, 10, 11$

If `schedule(static,4)` is used, the iterations will be assigned as

Thread 0: 0, 1, 2, 3

Thread 1: 4, 5, 6, 7

Thread 2: 8, 9, 10, 11

Thus the clause `schedule(static, total_iterations/thread_count)` is more or less equivalent to the default schedule used by most implementations of OpenMP.

The `chunksize` can be omitted. If it is omitted, the `chunksize` is approximately `total_iterations/thread_count`.

5.7.3 The `dynamic` and `guided` schedule types

In a `dynamic` schedule, the iterations are also broken up into chunks of `chunksize` consecutive iterations. Each thread executes a chunk, and when a thread finishes a chunk, it requests another one from the run-time system. This continues until all the iterations are completed. The `chunksize` can be omitted. When it is omitted, a `chunksize` of 1 is used.

In a `guided` schedule, each thread also executes a chunk, and when a thread finishes a chunk, it requests another one. However, in a `guided` schedule, as chunks are completed, the size of the new chunks decreases. For example, on one of our systems, if we run the trapezoidal rule program with the `parallel for` directive and a `schedule(guided)` clause, then when $n = 10{,}000$ and `thread_count` $= 2$, the iterations are assigned as shown in Table 5.3. We see that the size of the chunk is approximately the number of iterations remaining divided by the number of threads. The first chunk has size $9999/2 \approx 5000$, since there are 9999 unassigned iterations. The second chunk has size $4999/2 \approx 2500$, and so on.

In a `guided` schedule, if no `chunksize` is specified, the size of the chunks decreases down to 1. If `chunksize` is specified, it decreases down to `chunksize`, with the exception that the very last chunk can be smaller than `chunksize`.

5.7.4 The `runtime` schedule type

To understand `schedule(runtime)` we need to digress for a moment and talk about **environment variables**. As the name suggests, environment variables are named values that can be accessed by a running program. That is, they're available in the program's *environment*. Some commonly used environment variables are `PATH`, `HOME`, and `SHELL`. The `PATH` variable specifies which directories the shell should search when it's looking for an executable. It's usually defined in both Unix and Windows. The `HOME` variable specifies the location of the user's home directory, and the `SHELL` variable specifies the location of the executable for the user's shell. These are usually defined in Unix systems. In both Unix-like systems (e.g., Linux and Mac OS X) and Windows, environment variables can be examined and specified on the command line. In Unix-like systems, you can use the shell's command line.

Table 5.3 Assignment of Trapezoidal Rule Iterations 1–9999 using a `guided` Schedule with Two Threads

Thread	Chunk	Size of Chunk	Remaining Iterations
0	1–5000	5000	4999
1	5001–7500	2500	2499
1	7501–8750	1250	1249
1	8751–9375	625	624
0	9376–9687	312	312
1	9688–9843	156	156
0	9844–9921	78	78
1	9922–9960	39	39
1	9961–9980	20	19
1	9981–9990	10	9
1	9991–9995	5	4
0	9996–9997	2	2
1	9998–9998	1	1
0	9999–9999	1	0

In Windows systems, you can use the command line in an integrated development environment.

As an example, if we're using the bash shell, we can examine the value of an environment variable by typing

```
$ echo $PATH
```

and we can use the `export` command to set the value of an environment variable

```
$ export TEST_VAR="hello"
```

For details about how to examine and set environment variables for your particular system, you should consult with your local expert.

When `schedule(runtime)` is specified, the system uses the environment variable `OMP_SCHEDULE` to determine at run-time how to schedule the loop. The `OMP_SCHEDULE` environment variable can take on any of the values that can be used for a static, dynamic, or guided schedule. For example, suppose we have a `parallel for` directive in a program and it has been modified by `schedule(runtime)`. Then if we use the bash shell, we can get a cyclic assignment of iterations to threads by executing the command

```
$ export OMP_SCHEDULE="static,1"
```

Now, when we start executing our program, the system will schedule the iterations of the `for` loop as if we had the clause `schedule(static,1)` modifiying the `parallel for` directive.

5.7.5 Which schedule?

If we have a `for` loop that we're able to parallelize, how do we decide which type of schedule we should use and what the `chunksize` should be? As you may have guessed, there *is* some overhead associated with the use of a `schedule` clause. Furthermore, the overhead is greater for `dynamic` schedules than `static` schedules, and the overhead associated with `guided` schedules is the greatest of the three. Thus, if we're getting satisfactory performance without a `schedule` clause, we should go no further. However, if we suspect that the performance of the default schedule can be substantially improved, we should probably experiment with some different schedules.

In the example at the beginning of this section, when we switched from the default schedule to `schedule(static,1)`, the speedup of the two-threaded execution of the program increased from 1.33 to 1.99. Since it's *extremely* unlikely that we'll get speedups that are significantly better than 1.99, we can just stop here, at least if we're only going to use two threads with 10,000 iterations. If we're going to be using varying numbers of threads and varying numbers of iterations, we need to do more experimentation, and it's entirely possible that we'll find that the optimal schedule depends on both the number of threads and the number of iterations.

It can also happen that we'll decide that the performance of the default schedule isn't very good, and we'll proceed to search through a large array of schedules and iteration counts only to conclude that our loop doesn't parallelize very well and *no* schedule is going to give us much improved performance. For an example of this, see Programming Assignment 5.4.

There are some situations in which it's a good idea to explore some schedules before others:

- If each iteration of the loop requires roughly the same amount of computation, then it's likely that the default distribution will give the best performance.
- If the cost of the iterations decreases (or increases) linearly as the loop executes, then a `static` schedule with small chunksizes will probably give the best performance.
- If the cost of each iteration can't be determined in advance, then it may make sense to explore a variety of scheduling options. The `schedule(runtime)` clause can be used here, and the different options can be explored by running the program with different assignments to the environment variable `OMP_SCHEDULE`.

5.8 PRODUCERS AND CONSUMERS

Let's take a look at a parallel problem that isn't amenable to parallelization using a `parallel for` or `for` directive.

5.8.1 Queues

Recall that a **queue** is a list abstract datatype in which new elements are inserted at the "rear" of the queue and elements are removed from the "front" of the queue.

A queue can thus be viewed as an abstraction of a line of customers waiting to pay for their groceries in a supermarket. The elements of the list are the customers. New customers go to the end or "rear" of the line, and the next customer to check out is the customer standing at the "front" of the line.

When a new entry is added to the rear of a queue, we sometimes say that the entry has been "enqueued," and when an entry is removed from the front of a queue, we sometimes say that the entry has been "dequeued."

Queues occur frequently in computer science. For example, if we have a number of processes, each of which wants to store some data on a hard drive, then a natural way to insure that only one process writes to the disk at a time is to have the processes form a queue, that is, the first process that wants to write gets access to the drive first, the second process gets access to the drive next, and so on.

A queue is also a natural data structure to use in many multithreaded applications. For example, suppose we have several "producer" threads and several "consumer" threads. The producer threads might "produce" requests for data from a server—for example, current stock prices—while the consumer threads might "consume" the request by finding or generating the requested data—the current stock prices. The producer threads could enqueue the requested prices, and the consumer threads could dequeue them. In this example, the process wouldn't be completed until the consumer threads had given the requested data to the producer threads.

5.8.2 Message-passing

Another natural application would be implementing message-passing on a shared-memory system. Each thread could have a shared message queue, and when one thread wanted to "send a message" to another thread, it could enqueue the message in the destination thread's queue. A thread could receive a message by dequeuing the message at the head of its message queue.

Let's implement a relatively simple message-passing program in which each thread generates random integer "messages" and random destinations for the messages. After creating the message, the thread enqueues the message in the appropriate message queue. After sending a message, a thread checks its queue to see if it has received a message. If it has, it dequeues the first message in its queue and prints it out. Each thread alternates between sending and trying to receive messages. We'll let the user specify the number of messages each thread should send. When a thread is done sending messages, it receives messages until all the threads are done, at which point all the threads quit. Pseudocode for each thread might look something like this:

```
for (sent_msgs = 0; sent_msgs < send_max; sent_msgs++) {
   Send_msg();
   Try_receive();
}

while (!Done())
   Try_receive();
```

5.8.3 Sending messages

Note that accessing a message queue to enqueue a message is probably a critical section. Although we haven't looked into the details of the implementation of the message queue, it seems likely that we'll want to have a variable that keeps track of the rear of the queue. For example, if we use a singly linked list with the tail of the list corresponding to the rear of the queue, then, in order to efficiently enqueue, we would want to store a pointer to the rear. When we enqueue a new message, we'll need to check and update the rear pointer. If two threads try to do this simultaneously, we may lose a message that has been enqueued by one of the threads. (It might help to draw a picture!) The results of the two operations will conflict, and hence enqueueing a message will form a critical section.

Pseudocode for the `Send_msg()` function might look something like this:

```
    mesg = random();
    dest = random() % thread_count;
#   pragma omp critical
    Enqueue(queue, dest, my_rank, mesg);
```

Note that this allows a thread to send a message to itself.

5.8.4 Receiving messages

The synchronization issues for receiving a message are a little different. Only the owner of the queue (that is, the destination thread) will dequeue from a given message queue. As long as we dequeue one message at a time, if there are at least two messages in the queue, a call to `Dequeue` can't possibly conflict with any calls to `Enqueue`, so if we keep track of the size of the queue, we can avoid any synchronization (for example, `critical` directives), as long as there are at least two messages.

Now you may be thinking, "What about the variable storing the size of the queue?" This would be a problem if we simply store the size of the queue. However, if we store two variables, `enqueued` and `dequeued`, then the number of messages in the queue is

```
queue_size = enqueued - dequeued
```

and the only thread that will update `dequeued` is the owner of the queue. Observe that one thread can update `enqueued` at the same time that another thread is using it to compute `queue_size`. To see this, let's suppose thread q is computing `queue_size`. It will either get the old value of `enqueued` or the new value. It *may* therefore compute a `queue_size` of 0 or 1 when `queue_size` should actually be 1 or 2, respectively, but in our program this will only cause a modest delay. Thread q will try again later if `queue_size` is 0 when it should be 1, and it will execute the critical section directive unnecessarily if `queue_size` is 1 when it should be 2.

Thus, we can implement `Try_receive` as follows:

```
    queue_size = enqueued - dequeued;
    if (queue_size == 0) return;
```

```
      else if (queue_size == 1)
#        pragma omp critical
         Dequeue(queue, &src, &mesg);
      else
         Dequeue(queue, &src, &mesg);
      Print_message(src, mesg);
```

5.8.5 Termination detection

We also need to think about implementation of the `Done` function. First note that the following "obvious" implementation will have problems:

```
queue_size = enqueued - dequeued;
if (queue_size == 0)
   return TRUE;
else
   return FALSE;
```

If thread *u* executes this code, it's entirely possible that some thread—call it thread *v*—will send a message to thread *u after u* has computed `queue_size = 0`. Of course, after thread *u* computes `queue_size = 0`, it will terminate and the message sent by thread *v* will never be received.

However, in our program, after each thread has completed the `for` loop, it won't send any new messages. Thus, if we add a counter `done_sending`, and each thread increments this after completing its `for` loop, then we *can* implement `Done` as follows:

```
queue_size = enqueued - dequeued;
if (queue_size == 0 && done_sending == thread_count)
   return TRUE;
else
   return FALSE;
```

5.8.6 Startup

When the program begins execution, a single thread, the master thread, will get command-line arguments and allocate an array of message queues, one for each thread. This array needs to be shared among the threads, since any thread can send to any other thread, and hence any thread can enqueue a message in any of the queues. Given that a message queue will (at a minimum) store

- a list of messages,
- a pointer or index to the rear of the queue,
- a pointer or index to the front of the queue,
- a count of messages enqueued, and
- a count of messages dequeued,

it makes sense to store the queue in a struct, and in order to reduce the amount of copying when passing arguments, it also makes sense to make the message queue

an array of pointers to structs. Thus, once the array of queues is allocated by the master thread, we can start the threads using a `parallel` directive, and each thread can allocate storage for its individual queue.

An important point here is that one or more threads may finish allocating their queues before some other threads. If this happens, the threads that finish first could start trying to enqueue messages in a queue that hasn't been allocated and cause the program to crash. We therefore need to make sure that none of the threads starts sending messages until all the queues are allocated. Recall that we've seen that several OpenMP directives provide implicit barriers when they're completed, that is, no thread will proceed past the end of the block until all the threads in the team have completed the block. In this case, though, we'll be in the middle of a `parallel` block, so we can't rely on an implicit barrier from some other OpenMP construct—we need an *explicit* barrier. Fortunately, OpenMP provides one:

```
# pragma omp barrier
```

When a thread encounters the barrier, it blocks until all the threads in the team have reached the barrier. After all the threads have reached the barrier, all the threads in the team can proceed.

5.8.7 The `atomic` directive

After completing its sends, each thread increments `done_sending` before proceeding to its final loop of receives. Clearly, incrementing `done_sending` is a critical section, and we could protect it with a `critical` directive. However, OpenMP provides a potentially higher performance directive: the `atomic` directive:

```
# pragma omp atomic
```

Unlike the `critical` directive, it can only protect critical sections that consist of a single C assignment statement. Further, the statement must have one of the following forms:

```
x <op>= <expression>;
x++;
++x;
x—;
—x;
```

Here `<op>` can be one of the binary operators

```
+, *, −, /, &, ^, |, <<, or >>.
```

It's also important to remember that `<expression>` must not reference `x`.

It should be noted that only the load and store of `x` are guaranteed to be protected. For example, in the code

```
#     pragma omp atomic
      x += y++;
```

a thread's update to x will be completed before any other thread can begin updating x. However, the update to y may be unprotected and the results may be unpredictable.

The idea behind the atomic directive is that many processors provide a special load-modify-store instruction, and a critical section that only does a load-modify-store can be protected much more efficiently by using this special instruction rather than the constructs that are used to protect more general critical sections.

5.8.8 Critical sections and locks

To finish our discussion of the message-passing program, we need to take a more careful look at OpenMP's specification of the critical directive. In our earlier examples, our programs had at most one critical section, and the critical directive forced mutually exclusive access to the section by all the threads. In this program, however, the use of critical sections is more complex. If we simply look at the source code, we'll see three blocks of code preceded by a critical or an atomic directive:

- `done_sending++;`
- `Enqueue(q_p, my_rank, mesg);`
- `Dequeue(q_p, &src, &mesg);`

However, we don't need to enforce exclusive access across all three of these blocks of code. We don't even need to enforce completely exclusive access within the second and third blocks. For example, it would be fine for, say, thread 0 to enqueue a message in thread 1's queue at the same time that thread 1 is enqueuing a message in thread 2's queue. But for the second and third blocks—the blocks protected by critical directives—this is exactly what OpenMP does. From OpenMP's point of view our program has two distinct critical sections: the critical section protected by the atomic directive, done_sending++, and the "composite" critical section in which we enqueue and dequeue messages.

Since enforcing mutual exclusion among threads serializes execution, this default behavior of OpenMP—treating all critical blocks as part of one composite critical section—can be highly detrimental to our program's performance. OpenMP *does* provide the option of adding a name to a critical directive:

```
# pragma omp critical(name)
```

When we do this, two blocks protected with critical directives with different names *can* be executed simultaneously. However, the names are set during compilation, and we want a different critical section for each thread's queue. Therefore, we need to set the names at run-time, and in our setting, when we want to allow simultaneous access to the same block of code by threads accessing different queues, the named critical directive isn't sufficient.

The alternative is to use **locks**.[4] A lock consists of a data structure and functions that allow the programmer to explicitly enforce mutual exclusion in a critical section. The use of a lock can be roughly described by the following pseudocode:

```
/* Executed by one thread */
Initialize the lock data structure;
. . .
/* Executed by multiple threads */
Attempt to lock or set the lock data structure;
Critical section;
Unlock or unset the lock data structure;
. . .
/* Executed by one thread */
Destroy the lock data structure;
```

The lock data structure is shared among the threads that will execute the critical section. One of the threads (e.g., the master thread) will initialize the lock, and when all the threads are done using the lock, one of the threads should destroy it.

Before a thread enters the critical section, it attempts to *set* or lock the lock data structure by calling the lock function. If no other thread is executing code in the critical section, it *obtains* the lock and proceeds into the critical section past the call to the lock function. When the thread finishes the code in the critical section, it calls an unlock function, which *relinquishes* or *unsets* the lock and allows another thread to obtain the lock.

While a thread owns the lock, no other thread can enter the critical section. If another thread attempts to enter the critical section, it will *block* when it calls the lock function. If multiple threads are blocked in a call to the lock function, then when the thread in the critical section relinquishes the lock, one of the blocked threads returns from the call to the lock, and the others remain blocked.

OpenMP has two types of locks: **simple** locks and **nested** locks. A simple lock can only be set once before it is unset, while a nested lock can be set multiple times by the same thread before it is unset. The type of an OpenMP simple lock is `omp_lock_t`, and the simple lock functions that we'll be using are

```
void omp_init_lock(omp_lock_t*    lock_p   /* out */);
void omp_set_lock(omp_lock_t*     lock_p   /* in/out */);
void omp_unset_lock(omp_lock_t*   lock_p   /* in/out */);
void omp_destroy_lock(omp_lock_t* lock_p   /* in/out */);
```

The type and the functions are specified in `omp.h`. The first function initializes the lock so that it's unlocked, that is, no thread owns the lock. The second function attempts to set the lock. If it succeeds, the calling thread proceeds; if it fails, the calling thread blocks until the lock becomes available. The third function unsets the

[4] If you've studied the Pthreads chapter, you've already learned about locks, and you can skip ahead to the syntax for OpenMP locks.

lock so another thread can obtain it. The fourth function makes the lock uninitialized. We'll only use simple locks. For information about nested locks, see [8, 10], or [42].

5.8.9 Using locks in the message-passing program

In our earlier discussion of the limitations of the `critical` directive, we saw that in the message-passing program, we wanted to insure mutual exclusion in each individual message queue, not in a particular block of source code. Locks allow us to do this. If we include a data member with type `omp_lock_t` in our queue struct, we can simply call `omp_set_lock` each time we want to insure exclusive access to a message queue. So the code

```
#  pragma omp critical
   /* q_p = msg_queues[dest] */
   Enqueue(q_p, my_rank, mesg);
```

can be replaced with

```
/* q_p = msg_queues[dest] */
omp_set_lock(&q_p->lock);
Enqueue(q_p, my_rank, mesg);
omp_unset_lock(&q_p->lock);
```

Similarly, the code

```
#  pragma omp critical
   /* q_p = msg_queues[my_rank] */
   Dequeue(q_p, &src, &mesg);
```

can be replaced with

```
/* q_p = msg_queues[my_rank] */
omp_set_lock(&q_p->lock);
Dequeue(q_p, &src, &mesg);
omp_unset_lock(&q_p->lock);
```

Now when a thread tries to send or receive a message, it can only be blocked by a thread attempting to access the same message queue, since different message queues have different locks. In our original implementation, only one thread could send at a time, regardless of the destination.

Note that it would also be possible to put the calls to the lock functions in the queue functions `Enqueue` and `Dequeue`. However, in order to preserve the performance of `Dequeue`, we would also need to move the code that determines the size of the queue (`enqueued – dequeued`) to `Dequeue`. Without it, the `Dequeue` function will lock the queue every time it is called by `Try_receive`. In the interest of preserving the structure of the code we've already written, we'll leave the calls to `omp_set_lock` and `omp_unset_lock` in the `Send` and `Try_receive` functions.

Since we're now including the lock associated with a queue in the queue struct, we can add initialization of the lock to the function that initializes an empty queue.

Destruction of the lock can be done by the thread that owns the queue before it frees the queue.

5.8.10 `critical` directives, `atomic` directives, or locks?

Now that we have three mechanisms for enforcing mutual exclusion in a critical section, it's natural to wonder when one method is preferable to another. In general, the `atomic` directive has the potential to be the fastest method of obtaining mutual exclusion. Thus, if your critical section consists of an assignment statement having the required form, it will probably perform at least as well with the `atomic` directive as the other methods. However, the OpenMP specification [42] allows the `atomic` directive to enforce mutual exclusion across *all* `atomic` directives in the program—this is the way the unnamed `critical` directive behaves. If this might be a problem—for example, you have multiple different critical sections protected by `atomic` directives—you should use named `critical` directives or locks. For example, suppose we have a program in which it's possible that one thread will execute the code on the left while another executes the code on the right.

```
# pragma omp atomic          # pragma omp atomic
  x++;                         y++;
```

Even if `x` and `y` are unrelated memory locations, it's possible that if one thread is executing `x++`, then no thread can simultaneously execute `y++`. It's important to note that the standard doesn't require this behavior. If two statements are protected by `atomic` directives and the two statements modify different variables, then there are implementations that treat the two statements as different critical sections. See Exercise 5.10. On the other hand, different statements that modify the same variable *will* be treated as if they belong to the same critical section, regardless of the implementation.

We've already seen some limitations to the use of `critical` directives. However, both named and unnamed `critical` directives are very easy to use. Furthermore, in the implementations of OpenMP that we've used there doesn't seem to be a very large difference between the performance of critical sections protected by a critical directive, and `critical` sections protected by locks, so if you can't use an `atomic` directive, but you can use a `critical` directive, you probably should. Thus, the use of locks should probably be reserved for situations in which mutual exclusion is needed for a data structure rather than a block of code.

5.8.11 Some caveats

You should exercise caution when you're using the mutual exclusion techniques we've discussed. They can definitely cause serious programming problems. Here are a few things to be aware of:

1. You shouldn't mix the different types of mutual exclusion for a single critical section. For example, suppose a program contains the following two segments.

```
# pragma omp atomic          # pragma omp critical
  x += f(y);                   x = g(x);
```

The update to x on the right doesn't have the form required by the atomic directive, so the programmer used a critical directive. However, the critical directive won't exclude the action executed by the atomic block, and it's possible that the results will be incorrect. The programmer needs to either rewrite the function *g* so that its use can have the form required by the atomic directive, or she needs to protect both blocks with a critical directive.

2. There is no guarantee of **fairness** in mutual exclusion constructs. This means that it's possible that a thread can be blocked forever in waiting for access to a critical section. For example, in the code

```
   while(1) {
      . . .
#     pragma omp critical
      x = g(my_rank);
      . . .
   }
```

it's possible that, for example, thread 1 can block forever waiting to execute x = g(my_rank), while the other threads repeatedly execute the assignment. Of course, this wouldn't be an issue if the loop terminated. Also note that many implementations give threads access to the critical section in the order in which they reach it, and for these implementations, this won't be an issue.

3. It can be dangerous to "nest" mutual exclusion constructs. As an example, suppose a program contains the following two segments.

```
#  pragma omp critical
   y = f(x);
   . . .
   double f(double x) {
#     pragma omp critical
      z = g(x);  /* z is shared */
      . . .
   }
```

This is guaranteed to **deadlock**. When a thread attempts to enter the second critical section, it will block forever. If thread *u* is executing code in the first critical block, no thread can execute code in the second block. In particular, thread *u* can't execute this code. However, if thread *u* is blocked waiting to enter the second critical block, then it will never leave the first, and it will stay blocked forever.

In this example, we can solve the problem by using named critical sections. That is, we could rewrite the code as

```
#  pragma omp critical(one)
   y = f(x);
   . . .
   double f(double x) {
#     pragma omp critical(two)
      z = g(x);  /* z is global */
      . . .
   }
```

However, it's not difficult to come up with examples when naming won't help. For example, if a program has two named critical sections—say `one` and `two`—and threads can attempt to enter the critical sections in different orders, then deadlock can occur. For example, suppose thread *u* enters `one` at the same time that thread *v* enters `two` and *u* then attempts to enter `two` while *v* attempts to enter `one`:

Time	Thread *u*	Thread *v*
0	Enter crit. sect. `one`	Enter crit. sect. `two`
1	Attempt to enter `two`	Attempt to enter `one`
2	Block	Block

Then both *u* and *v* will block forever waiting to enter the critical sections. So it's not enough to just use different names for the critical sections—the programmer must insure that different critical sections are always entered in the same order.

5.9 CACHES, CACHE COHERENCE, AND FALSE SHARING[5]

Recall that for a number of years now, processors have been able to execute operations much faster than they can access data in main memory, so if a processor must read data from main memory for each operation, it will spend most of its time simply waiting for the data from memory to arrive. Also recall that in order to address this problem, chip designers have added blocks of relatively fast memory to processors. This faster memory is called **cache memory**.

The design of cache memory takes into consideration the principles of *temporal and spatial locality*: if a processor accesses main memory location *x* at time *t*, then it is likely that at times close to *t*, it will access main memory locations close to *x*. Thus, if a processor needs to access main memory location *x*, rather than transferring only the contents of *x* to/from main memory, a block of memory containing *x* is tranferred from/to the processor's cache. Such a block of memory is called a **cache line** or **cache block**.

We've already seen in Section 2.3.4 that the use of cache memory can have a huge impact on shared memory. Let's recall why. First, consider the following situation. Suppose `x` is a shared variable with the value 5, and both thread 0 and thread 1 read `x` from memory into their (separate) caches, because both want to execute the statement

```
my_y = x;
```

[5]This material is also covered in Chapter 4. So if you've already read that chapter, you may want to just skim this section.

Here, `my_y` is a private variable defined by both threads. Now suppose thread 0 executes the statement

```
x++;
```

Finally, suppose that thread 1 now executes

```
my_z = x;
```

where `my_z` is another private variable.

What's the value in `my_z`? Is it 5? Or is it 6? The problem is that there are (at least) three copies of `x`: the one in main memory, the one in thread 0's cache, and the one in thread 1's cache. When thread 0 executed `x++`, what happened to the values in main memory and thread 1's cache? This is the **cache coherence** problem, which we discussed in Chapter 2. We saw there that most systems insist that the caches be made aware that changes have been made to data they are caching. The line in the cache of thread 1 would have been marked *invalid* when thread 0 executed `x++`, and before assigning `my_z = x`, the core running thread 1 would see that it's value of `x` was out of date. Thus, the core running thread 0 would have to update the copy of `x` in main memory (either now or earlier), and the core running thread 1 would get the line with the updated value of `x` from main memory. For further details, see Chapter 2.

The use of cache coherence can have a dramatic effect on the performance of shared-memory systems. To illustrate this, let's take a look at matrix-vector multiplication. Recall that if $A = (a_{ij})$ is an $m \times n$ matrix and $\mathbf{x}$ is a vector with n components, then their product $\mathbf{y} = A\mathbf{x}$ is a vector with m components, and its ith component y_i is found by forming the dot product of the ith row of A with $\mathbf{x}$:

$$y_i = a_{i0}x_0 + a_{i1}x_1 + \cdots + a_{i,n-1}x_{n-1}.$$

See Figure 5.5.

So if we store A as a two-dimensional array and $\mathbf{x}$ and $\mathbf{y}$ as one-dimensional arrays, we can implement serial matrix-vector multiplication with the following code:

```
for (i = 0; i < m; i++) {
   y[i] = 0.0;
```

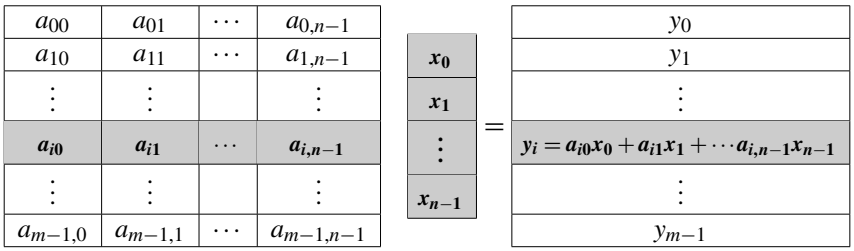


FIGURE 5.5

Matrix-vector multiplication

```
      for (j = 0; j < n; j++)
         y[i] += A[i][j]*x[j];
   }
```

There are no loop-carried dependences in the outer loop, since `A` and `x` are never updated and iteration i only updates `y[i]`. Thus, we can parallelize this by dividing the iterations in the outer loop among the threads:

```
1  #  pragma omp parallel for num_threads(thread_count)  \
2        default(none) private(i, j) shared(A, x, y, m, n)
3     for (i = 0; i < m; i++) {
4        y[i] = 0.0;
5        for (j = 0; j < n; j++)
6           y[i] += A[i][j]*x[j];
7     }
```

If T_{serial} is the run-time of the serial program and T_{parallel} is the run-time of the parallel program, recall that the *efficiency* E of the parallel program is the speedup S divided by the number of threads, t:

$$E = \frac{S}{t} = \frac{\left(\frac{T_{\text{serial}}}{T_{\text{parallel}}}\right)}{t} = \frac{T_{\text{serial}}}{t \times T_{\text{parallel}}}.$$

Since $S \leq t$, $E \leq 1$. Table 5.4 shows the run-times and efficiencies of our matrix-vector multiplication with different sets of data and differing numbers of threads.

In each case, the total number of floating point additions and multiplications is $64,000,000$. An analysis that only considers arithmetic operations would predict that a single thread running the code would take the same amount of time for all three inputs. However, it's clear that this is *not* the case. The $8,000,000 \times 8$ system requires about 22% more time than the 8000×8000 system, and the $8 \times 8,000,000$ system requires about 26% more time than the 8000×8000 system. Both of these differences are at least partially attributable to cache performance.

Recall that a **write-miss** occurs when a core tries to update a variable that's not in cache, and it has to access main memory. A cache profiler (such as Valgrind [49]) shows that when the program is run with the $8,000,000 \times 8$ input, it has far more

Table 5.4 Run-Times and Efficiencies of Matrix-Vector Multiplication (times in seconds)

	Matrix Dimension					
	$8,000,000 \times 8$		8000×8000		$8 \times 8,000,000$	
Threads	Time	Eff.	Time	Eff.	Time	Eff.
1	0.322	1.000	0.264	1.000	0.333	1.000
2	0.219	0.735	0.189	0.698	0.300	0.555
4	0.141	0.571	0.119	0.555	0.303	0.275

cache write-misses than either of the other inputs. The bulk of these occur in Line 4. Since the number of elements in the vector y is far greater in this case (8,000,000 vs. 8000 or 8), and each element must be initialized, it's not surprising that this line slows down the execution of the program with the $8{,}000{,}000 \times 8$ input.

Also recall that a **read-miss** occurs when a core tries to read a variable that's not in cache, and it has to access main memory. A cache profiler shows that when the program is run with the $8 \times 8{,}000{,}000$ input, it has far more cache read-misses than either of the other inputs. These occur in Line 6, and a careful study of this program (see Exercise 5.12) shows that the main source of the differences is due to the reads of x. Once again, this isn't surprising, since for this input, x has 8,000,000 elements, versus only 8000 or 8 for the other inputs.

It should be noted that there may be other factors that affect the relative performance of the single-threaded program with differing inputs. For example, we haven't taken into consideration whether virtual memory (see Section 2.2.4) has affected the performance of the program with the different inputs. How frequently does the CPU need to access the page table in main memory?

Of more interest to us, though, are the differences in efficiency as the number of threads is increased. The two-thread efficiency of the program with the $8 \times 8{,}000{,}000$ input is more than 20% less than the efficiency of the program with the $8{,}000{,}000 \times 8$ and the 8000×8000 inputs. The four-thread efficiency of the program with the $8 \times 8{,}000{,}000$ input is more than 50% less than the program's efficiency with the $8{,}000{,}000 \times 8$ and the 8000×8000 inputs. Why, then, is the multithreaded performance of the program so much worse with the $8 \times 8{,}000{,}000$ input?

In this case, once again, the answer has to do with cache. Let's take a look at the program when we run it with four threads. With the $8{,}000{,}000 \times 8$ input, y has 8,000,000 components, so each thread is assigned 2,000,000 components. With the 8000×8000 input, each thread is assigned 2000 components of y, and with the $8 \times 8{,}000{,}000$ input, each thread is assigned two components. On the system we used, a cache line is 64 bytes. Since the type of y is `double`, and a `double` is 8 bytes, a single cache line will store eight `double`s.

Cache coherence is enforced at the "cache-line level." That is, each time any value in a cache line is written, if the line is also stored in another core's cache, the entire *line* will be invalidated—not just the value that was written. The system we're using has two dual-core processors and each processor has its own cache. Suppose for the moment that threads 0 and 1 are assigned to one of the processors and threads 2 and 3 are assigned to the other. Also suppose that for the $8 \times 8{,}000{,}000$ problem all of y is stored in a single cache line. Then every write to some element of y will invalidate the line in the other processor's cache. For example, each time thread 0 updates `y[0]` in the statement

```
y[i] += A[i][j]*x[j];
```

if thread 2 or 3 is executing this code, it will have to reload y. Each thread will update each of its components 8,000,000 times. We see that with this assignment of threads to processors and components of y to cache lines, all the threads will

have to reload `y` *many* times. This is going to happen in spite of the fact that only one thread accesses any one component of `y`—for example, only thread 0 accesses `y[0]`.

Each thread will update its assigned components of `y` a total of 16,000,000 times. It appears that many, if not most, of these updates are forcing the threads to access main memory. This is called **false sharing**. Suppose two threads with separate caches access different variables that belong to the same cache line. Further suppose at least one of the threads updates its variable. Even though neither thread has written to a shared variable, the cache controller invalidates the entire cache line and forces the other threads to get the values of the variables from main memory. The threads aren't sharing anything (except a cache line), but the behavior of the threads with respect to memory access is the same as if they were sharing a variable. Hence the name *false sharing*.

Why is false sharing not a problem with the other inputs? Let's look at what happens with the 8000×8000 input. Suppose thread 2 is assigned to one of the processors and thread 3 is assigned to another. (We don't actually know which threads are assigned to which processors, but it turns out—see Exercise 5.13—that it doesn't matter.) Thread 2 is responsible for computing

```
y[4000], y[4001], . . . , y[5999],
```

and thread 3 is responsible for computing

```
y[6000], y[6001], . . . , y[7999]
```

If a cache line contains eight consecutive `doubles`, the only possibility for false sharing is on the interface between their assigned elements. If, for example, a single cache line contains

```
y[5996], y[5997], y[5998], y[5999], y[6000], y[6001], y[6002], y[6003],
```

then it's conceivable that there might be false sharing of this cache line. However, thread 2 will access

```
y[5996], y[5997], y[5998], y[5999]
```

at the *end* of its iterations of the `for i` loop, while thread 3 will access

```
y[6000], y[6001], y[6002], y[6003]
```

at the *beginning* of its iterations. So it's very likely that when thread 2 accesses, say, `y[5996]`, thread 3 will be long done with all four of

```
y[6000], y[6001], y[6002], y[6003].
```

Similarly, when thread 3 accesses, say, `y[6003]`, it's very likely that thread 2 won't be anywhere near starting to access

```
y[5996], y[5997], y[5998], y[5999].
```

It's therefore unlikely that false sharing of the elements of `y` will be a significant problem with the 8000×8000 input. Similar reasoning suggests that false sharing of `y` is unlikely to be a problem with the $8,000,000 \times 8$ input. Also note that we don't need to worry about false sharing of `A` or `x`, since their values are never updated by the matrix-vector multiplication code.

This brings up the question of how we might avoid false sharing in our matrix-vector multiplication program. One possible solution is to "pad" the `y` vector with dummy elements in order to insure that any update by one thread won't affect another thread's cache line. Another alternative is to have each thread use its own private storage during the multiplication loop, and then update the shared storage when they're done (see Exercise 5.15).

5.10 THREAD-SAFETY[6]

Let's look at another potential problem that occurs in shared-memory programming: *thread-safety.* A block of code is **thread-safe** if it can be simultaneously executed by multiple threads without causing problems.

As an example, suppose we want to use multiple threads to "tokenize" a file. Let's suppose that the file consists of ordinary English text, and that the tokens are just contiguous sequences of characters separated from the rest of the text by white space—spaces, tabs, or newlines. A simple approach to this problem is to divide the input file into lines of text and assign the lines to the threads in a round-robin fashion: the first line goes to thread 0, the second goes to thread 1, ..., the tth goes to thread t, the $t+1$st goes to thread 0, and so on.

We'll read the text into an array of strings, with one line of text per string. Then we can use a `parallel for` directive with a `schedule(static,1)` clause to divide the lines among the threads.

One way to tokenize a line is to use the `strtok` function in `string.h`. It has the following prototype:

```
char* strtok(
      char*        string      /* in/out */,
      const char*  separators  /* in     */);
```

Its usage is a little unusual: the first time it's called, the `string` argument should be the text to be tokenized, so in our example it should be the line of input. For subsequent calls, the first argument should be `NULL`. The idea is that in the first call, `strtok` caches a pointer to `string`, and for subsequent calls it returns successive tokens taken from the cached copy. The characters that delimit tokens should be passed in `separators`, so we should pass in the string `" \t\n"` as the `separators` argument.

[6]This material is also covered in Chapter 4, so if you've already read that chapter, you may want to just skim this section.

```
 1  void Tokenize(
 2        char*  lines[]       /* in/out */,
 3        int    line_count    /* in     */,
 4        int    thread_count  /* in     */) {
 5     int my_rank, i, j;
 6     char *my_token;
 7
 8  #  pragma omp parallel num_threads(thread_count) \
 9        default(none) private(my_rank, i, j, my_token) \
10        shared(lines, line_count)
11     {
12        my_rank = omp_get_thread_num();
13  #     pragma omp for schedule(static, 1)
14        for (i = 0; i < line_count; i++) {
15           printf("Thread %d > line %d = %s", my_rank, i,
                    lines[i]);
16           j = 0;
17           my_token = strtok(lines[i], " \t\n");
18           while ( my_token != NULL ) {
19              printf("Thread %d > token %d = %s\n", my_rank, j,
                       my_token);
20              my_token = strtok(NULL, " \t\n");
21              j++;
22           }
23        } /* for i */
24     }  /* omp parallel */
25
26  }  /* Tokenize */
```

Program 5.6: A first attempt at a multi threaded tokenizer

Given these assumptions, we can write the `Tokenize` function shown in Program 5.6. The main function has initialized the array `lines` so that it contains the input text, and `line_count` is the number of strings stored in `lines`. Although for our purposes, we only need the `lines` argument to be an input argument, the `strtok` function modifies its input. Thus, when `Tokenize` returns, `lines` will be modified. When we run the program with a single thread, it correctly tokenizes the input stream. The first time we run it with two threads and the input

Pease porridge hot.
Pease porridge cold.
Pease porridge in the pot
Nine days old.

the output is also correct. However, the second time we run it with this input, we get the following output.

```
Thread 0 > line 0 = Pease porridge hot.
Thread 1 > line 1 = Pease porridge cold.
Thread 0 > token 0 = Pease
```

```
Thread 1 > token 0 = Pease
Thread 0 > token 1 = porridge
Thread 1 > token 1 = cold.
Thread 0 > line 2 = Pease porridge in the pot
Thread 1 > line 3 = Nine days old.
Thread 0 > token 0 = Pease
Thread 1 > token 0 = Nine
Thread 0 > token 1 = days
Thread 1 > token 1 = old.
```

What happened? Recall that `strtok` caches the input line. It does this by declaring a variable to have `static` storage class. This causes the value stored in this variable to persist from one call to the next. Unfortunately for us, this cached string is shared, not private. Thus, it appears that thread 1's call to `strtok` with the second line has apparently overwritten the contents of thread 0's call with the first line. Even worse, thread 0 has found a token ("days") that should be in thread 1's output.

The `strtok` function is therefore *not* thread-safe: if multiple threads call it simultaneously, the output it produces may not be correct. Regrettably, it's not uncommon for C library functions to fail to be thread-safe. For example, neither the random number generator `random` in `stdlib.h` nor the time conversion function `localtime` in `time.h` is thread-safe. In some cases, the C standard specifies an alternate, thread-safe, version of a function. In fact, there is a thread-safe version of `strtok`:

```
char* strtok_r(
      char*        string       /* in/out */,
      const char*  separators   /* in     */,
      char**       saveptr_p    /* in/out */);
```

The "`_r`" is supposed to suggest that the function is *re-entrant*, which is sometimes used as a synonym for thread-safe. The first two arguments have the same purpose as the arguments to `strtok`. The `saveptr_p` argument is used by `strtok_r` for keeping track of where the function is in the input string; it serves the purpose of the cached pointer in `strtok`. We can correct our original `Tokenize` function by replacing the calls to `strtok` with calls to `strtok_r`. We simply need to declare a `char*` variable to pass in for the third argument, and replace the calls in Line 17 and Line 20 with the calls

```
my_token = strtok_r(lines[i], " \t\n", &saveptr);
. . .
my_token = strtok_r(NULL, " \t\n", &saveptr);
```

respectively.

5.10.1 Incorrect programs can produce correct output

Notice that our original version of the tokenizer program shows an especially insidious form of program error: The first time we ran it with two threads, the program produced correct output. It wasn't until a later run that we saw an error. This,

unfortunately, is not a rare occurrence in parallel programs. It's especially common in shared-memory programs. Since, for the most part, the threads are running independently of each other, as we noted back at the beginning of the chapter, the exact sequence of statements executed is nondeterministic. For example, we can't say when thread 1 will first call `strtok`. If its first call takes place after thread 0 has tokenized its first line, then the tokens identified for the first line should be correct. However, if thread 1 calls `strtok` before thread 0 has finished tokenizing its first line, it's entirely possible that thread 0 may not identify all the tokens in the first line, so it's especially important in developing shared-memory programs to resist the temptation to assume that since a program produces correct output, it must be correct. We always need to be wary of race conditions.

5.11 SUMMARY

OpenMP is a standard for programming shared-memory systems. It uses both special functions and preprocessor directives called **pragmas**, so unlike Pthreads and MPI, OpenMP requires compiler support. One of the most important features of OpenMP is that it was designed so that developers could *incrementally* parallelize existing serial programs, rather than having to write parallel programs from scratch.

OpenMP programs start multiple **threads** rather than multiple processes. Threads can be much lighter weight than processes; they can share almost all the resources of a process, except each thread must have its own stack and program counter.

To get OpenMP's function prototypes and macros, we include the `omp.h` header in OpenMP programs. There are several OpenMP directives that start multiple threads; the most general is the `parallel` directive:

```
#  pragma omp parallel
   structured block
```

This directive tells the run-time system to execute the following structured block of code in parallel. It may **fork** or start several threads to execute the structured block. A **structured block** is a block of code with a single entry point and a single exit point, although calls to the C library function `exit` are allowed within a structured block. The number of threads started is system dependent, but most systems will start one thread for each available core. The collection of threads executing `block of code` is called a **team**. One of the threads in the team is the thread that was executing the code before the `parallel` directive. This thread is called the **master**. The additional threads started by the `parallel` directive are called **slaves**. When all of the threads are finished, the slave threads are terminated or **joined** and the master thread continues executing the code beyond the structured block.

Many OpenMP directives can be modified by **clauses**. We made frequent use of the `num_threads` clause. When we use an OpenMP directive that starts a team of threads, we can modify it with the `num_threads` clause so that the directive will start the number of threads we desire.

When OpenMP starts a team of threads, each of the threads is assigned a rank or an id in the range $0, 1, \ldots,$ `thread_count` $- 1$. The OpenMP library function `omp_get_thread_num` then returns the calling thread's rank. The function `omp_get_num_threads` returns the number of threads in the current team.

A major problem in the development of shared-memory programs is the possibility of **race conditions**. A race condition occurs when multiple threads attempt to access a shared resource, at least one of the accesses is an update, and the accesses can result in an error. Code that is executed by multiple threads that update a shared resource that can only be updated by one thread at a time is called a **critical section**. Thus, if multiple threads try to update a shared variable, the program has a race condition and the code that updates the variable is a critical section. OpenMP provides several mechanisms for insuring **mutual exclusion** in critical sections. We examined four of them:

1. `Critical` directives insure that only one thread at a time can execute the structured block. If multiple threads try to execute the code in the critical section, all but one of them will block before the critical section. As threads finish the critical section, other threads will be unblocked and enter the code.
2. Named `critical` directives can be used in programs having different critical sections that can be executed concurrently. Multiple threads trying to execute code in critical section(s) with the same name will be handled in the same way as multiple threads trying to execute an unnamed critical section. However, threads entering critical sections with different names can execute concurrently.
3. An `atomic` directive can only be used when the critical section has the form `x <op>= <expression>`, `x++`, `++x`, `x--`, or `--x`. It's designed to exploit special hardware instructions, so it can be much faster than an ordinary critical section.
4. Simple locks are the most general form of mutual exclusion. They use function calls to restrict access to a critical section:

   ```
   omp_set_lock(&lock);
   critical section
   omp_unset_lock(&lock);
   ```

   When multiple threads call `omp_set_lock`, only one of them will proceed to the critical section. The others will block until the first thread calls `omp_unset_lock`. Then one of the blocked threads can proceed.

All of the mutual exclusion mechanisms can cause serious program problems such as deadlock, so they need to be used with great care.

A `for` directive can be used to partition the iterations in a `for` loop among the threads. This directive doesn't start a team of threads, it divides the iterations in a `for` loop among the threads in an existing team. If we want to also start a team of threads, we can use the `parallel for` directive. There are a number of restrictions on the form of a `for` loop that can be parallelized; basically, the run-time system must

be able to determine the total number of iterations through the loop body before the loop begins execution. For details, see Program 5.3.

It's not enough, however, to insure that our `for` loop has one of the canonical forms. It must also not have any **loop-carried dependences**. A loop-carried dependence occurs when a memory location is read or written in one iteration and written in another iteration. OpenMP won't detect loop-carried dependences; it's up to us, the programmers, to detect them and eliminate them. It may, however, be impossible to eliminate them, in which case, the loop isn't a candidate for parallelization.

By default, most systems use a **block partitioning** of the iterations in a parallelized `for` loop. If there are n iterations, this means that roughly the first n/`thread_count` are assigned to thread 0, the next n/`thread_count` are assigned to thread 1, and so on. However, there are a variety of **scheduling** options provided by OpenMP. The `schedule` clause has the form

```
schedule(<type> [,<chunksize>])
```

The `type` can be `static`, `dynamic`, `guided`, `auto`, or `runtime`. In a `static` schedule, the iterations can be assigned to the threads before the loop starts execution. In `dynamic` and `guided` schedules the iterations are assigned on the fly. When a thread finishes a chunk of iterations—a contiguous block of iterations—it requests another chunk. If `auto` is specified, the schedule is determined by the compiler or run-time system, and if `runtime` is specified, the schedule is determined at run-time by examining the environment variable `OMP_SCHEDULE`.

Only `static`, `dynamic`, and `guided` schedules can have a `chunksize`. In a `static` schedule, the chunks of `chunksize` iterations are assigned in round robin fashion to the threads. In a `dynamic` schedule, each thread is assigned `chunksize` iterations, and when a thread completes its chunk, it requests another chunk. In a `guided` schedule, the size of the chunks decreases as the iteration proceeds.

In OpenMP the **scope** of a variable is the collection of threads to which the variable is accessible. Typically, any variable that was defined before the OpenMP directive has **shared** scope within the construct. That is, all the threads have access to it. The principal exception to this is that the loop variable in a `for` or `parallel for` construct is **private**, that is, each thread has its own copy of the variable. Variables that are defined within an OpenMP construct have private scope, since they will be allocated from the executing thread's stack.

As a rule of thumb, it's a good idea to explicitly assign the scope of variables. This can be done by modifying a `parallel` or `parallel for` directive with the *scoping* clause:

```
default(none)
```

This tells the system that the scope of every variable that's used in the OpenMP construct must be explicitly specified. Most of the time this can be done with `private` or `shared` clauses.

The only exceptions we encountered were **reduction variables**. A **reduction operator** is a binary operation (such as addition or multiplication) and a **reduction** is a computation that repeatedly applies the same reduction operator to a sequence of operands in order to get a single result. Furthermore, all of the intermediate results of the operation should be stored in the same variable: the **reduction variable**. For example, if A is an array with n elements, then the code

```
int sum = 0;
for (i = 0; i < n; i++)
   sum += A[i];
```

is a reduction. The reduction operator is addition and the reduction variable is `sum`. If we try to parallelize this loop, the reduction variable should have properties of both private and shared variables. Initially we would like each thread to add its array elements into its own private `sum`, but when the threads are done, we want the private `sum`'s combined into a single, shared `sum`. OpenMP therefore provides the `reduction` clause for identifying reduction variables and operators.

A `barrier` directive will cause the threads in a team to block until all the threads have reached the directive. We've seen that the `parallel`, `parallel for`, and `for` directives have implicit barriers at the end of the structured block.

We recalled that modern microprocessor architectures use caches to reduce memory access times, so typical architectures have special hardware to insure that the caches on the different chips are **coherent**. Since the unit of cache coherence, a **cache line** or **cache block**, is usually larger than a single word of memory, this can have the unfortunate side effect that two threads may be accessing different memory locations, but when the two locations belong to the same cache line, the cache-coherence hardware acts as if the threads were accessing the same memory location—if one of the threads updates its memory location, and then the other thread tries to read its memory location, it will have to retrieve the value from main memory. That is, the hardware is forcing the thread to act as if it were actually sharing the memory location. Hence, this is called **false sharing**, and it can seriously degrade the performance of a shared-memory program.

Some C functions cache data between calls by declaring variables to be `static`. This can cause errors when multiple threads call the function; since static storage is shared among the threads, one thread can overwrite another thread's data. Such a function is not **thread-safe**, and, unfortunately, there are several such functions in the C library. Sometimes, however, the library has a thread-safe variant of a function that isn't thread-safe.

In one of our programs we saw a particularly insidious problem: when we ran the program with multiple threads and a fixed set of input, it sometimes produced correct output, even though it had an error. Producing correct output during testing doesn't guarantee that the program is in fact correct. It's up to us to identify possible race condtions.

5.12 EXERCISES

5.1. If it's defined, the `_OPENMP` macro is a decimal `int`. Write a program that prints its value. What is the significance of the value?

5.2. Download `omp_trap_1.c` from the book's website, and delete the `critical` directive. Now compile and run the program with more and more threads and larger and larger values of n. How many threads and how many trapezoids does it take before the result is incorrect?

5.3. Modify `omp_trap_1.c` so that

a. it uses the first block of code on page 222, and

b. the time used by the `parallel` block is timed using the OpenMP function `omp_get_wtime()`. The syntax is

```
double omp_get_wtime(void)
```

It returns the number of seconds that have passed since some time in the past. For details on taking timings, see Section 2.6.4. Also recall that OpenMP has a `barrier` directive:

```
#  pragma omp barrier
```

Now find a system with at least two cores and time the program with

c. one thread and a large value of n, and

d. two threads and the same value of n.

What happens? Download `omp_trap_2.c` from the book's website. How does its performance compare? Explain your answers.

5.4. Recall that OpenMP creates private variables for reduction variables, and these private variables are initialized to the identity value for the reduction operator. For example, if the operator is addition, the private variables are initialized to 0, while if the operator is multiplication, the private variables are initialized to 1. What are the identity values for the operators `&&`, `||`, `&`, `|`, `^`?

5.5. Suppose that on the amazing Bleeblon computer, variables with type `float` can store three decimal digits. Also suppose that the Bleeblon's floating point registers can store four decimal digits, and that after any floating point operation, the result is rounded to three decimal digits before being stored. Now suppose a C program declares an array `a` as follows:

```
float a[] = {4.0, 3.0, 3.0, 1000.0};
```

a. What is the output of the following block of code if it's run on the Bleeblon?

```
int i;
float sum = 0.0;
```

```
for (i = 0; i < 4; i++)
   sum += a[i];
printf("sum = %4.1f\n", sum);
```

b. Now consider the following code:

```
   int i;
   float sum = 0.0;
#  pragma omp parallel for num_threads(2) \
      reduction(+:sum)
   for (i = 0; i < 4; i++)
      sum += a[i];
   printf("sum = %4.1f\n", sum);
```

Suppose that the run-time system assigns iterations `i = 0, 1` to thread 0 and `i = 2, 3` to thread 1. What is the output of this code on the Bleeblon?

5.6. Write an OpenMP program that determines the default scheduling of parallel `for` loops. Its input should be the number of iterations, and its output should be which iterations of a parallelized `for` loop are executed by which thread. For example, if there are two threads and four iterations, the output might be:

```
Thread 0: Iterations 0 — 1
Thread 1: Iterations 2 — 3
```

5.7. In our first attempt to parallelize the program for estimating π, our program was incorrect. In fact, we used the result of the program when it was run with one thread as evidence that the program run with two threads was incorrect. Explain why we could "trust" the result of the program when it was run with one thread.

5.8. Consider the loop

```
a[0] = 0;
for (i = 1; i < n; i++)
   a[i] = a[i-1] + i;
```

There's clearly a loop-carried dependence, as the value of `a[i]` can't be computed without the value of `a[i-1]`. Can you see a way to eliminate this dependence and parallelize the loop?

5.9. Modify the trapezoidal rule program that uses a `parallel for` directive (`omp_trap_3.c`) so that the `parallel for` is modified by a `schedule(runtime)` clause. Run the program with various assignments to the environment variable `OMP_SCHEDULE` and determine which iterations are assigned to which thread. This can be done by allocating an array `iterations` of *n* `int`s and in the `Trap` function assigning `omp_get_thread_num()` to `iterations[i]` in the *i*th iteration of the `for` loop. What is the default assignment of iterations on your system? How are `guided` schedules determined?

5.10. Recall that all structured blocks modified by an unnamed `critical` directive form a single critical section. What happens if we have a number of `atomic` directives in which different variables are being modified? Are they all treated as a single critical section?

We can write a small program that tries to determine this. The idea is to have all the threads simultaneously execute something like the following code

```
   int i;
   double my_sum = 0.0;
   for (i = 0; i < n; i++)
#     pragma omp atomic
      my_sum += sin(i);
```

We can do this by modifying the code by a `parallel` directive:

```
#  pragma omp parallel num_threads(thread_count)
   {
      int i;
      double my_sum = 0.0;
      for (i = 0; i < n; i++)
#        pragma omp atomic
         my_sum += sin(i);
   }
```

Note that since `my_sum` and `i` are declared in the `parallel` block, each thread has its own private copy. Now if we time this code for large n when `thread_count` $= 1$ and we also time it when `thread_count` > 1, then as long as `thread_count` is less than the number of available cores, the run-time for the single-threaded run should be roughly the same as the time for the multithreaded run if the different threads' executions of `my_sum += sin(i)` are treated as different critical sections. On the other hand, if the different executions of `my_sum += sin(i)` are all treated as a single critical section, the multithreaded run should be much slower than the single-threaded run. Write an OpenMP program that implements this test. Does your implementation of OpenMP allow simultaneous execution of updates to different variables when the updates are protected by `atomic` directives?

5.11. Recall that in C, a function that takes a two-dimensional array argument must specify the number of columns in the argument list, so it is quite common for C programmers to only use one-dimensional arrays, and to write explicit code for converting pairs of subscripts into a single dimension. Modify the OpenMP matrix-vector multiplication so that it uses a one-dimensional array for the matrix.

5.12. Download the source file `omp_mat_vect_rand_split.c` from the book's website. Find a program that does cache profiling (e.g., Valgrind [49]) and compile the program according to the instructions in the cache profiler documentation. (For example, with Valgrind you will want a symbol table and full optimization. (With `gcc` use, `gcc -g -O2 . . .`). Now run the program according to the instructions in the cache profiler documentation, using input $k \times (k \cdot 10^6)$,

$(k \cdot 10^3) \times (k \cdot 10^3)$, and $(k \cdot 10^6) \times k$. Choose k so large that the number of level 2 cache misses is of the order 10^6 for at least one of the input sets of data.

a. How many level 1 cache write-misses occur with each of the three inputs?
b. How many level 2 cache write-misses occur with each of the three inputs?
c. Where do most of the write-misses occur? For which input data does the program have the most write-misses? Can you explain why?
d. How many level 1 cache read-misses occur with each of the three inputs?
e. How many level 2 cache read-misses occur with each of the three inputs?
f. Where do most of the read-misses occur? For which input data does the program have the most read-misses? Can you explain why?
g. Run the program with each of the three inputs, but without using the cache profiler. With which input is the program the fastest? With which input is the program the slowest? Can your observations about cache misses help explain the differences? How?

5.13. Recall the matrix-vector multiplication example with the 8000×8000 input. Suppose that thread 0 and thread 2 are assigned to different processors. If a cache line contains 64 bytes or 8 `doubles`, is it possible for false sharing between threads 0 and 2 to occur for any part of the vector `y`? Why? What about if thread 0 and thread 3 are assigned to different processors; is it possible for false sharing to occur between them for any part of `y`?

5.14. Recall the matrix-vector multiplication example with an $8 \times 8{,}000{,}000$ matrix. Suppose that `doubles` use 8 bytes of memory and that a cache line is 64 bytes. Also suppose that our system consists of two dual-core processors.

a. What is the minimum number of cache lines that are needed to store the vector `y`?
b. What is the maximum number of cache lines that are needed to store the vector `y`?
c. If the boundaries of cache lines always coincide with the boundaries of 8-byte `doubles`, in how many different ways can the components of `y` be assigned to cache lines?
d. If we only consider which pairs of threads share a processor, in how many different ways can four threads be assigned to the processors in our computer? Here, we're assuming that cores on the same processor share cache.
e. Is there an assignment of components to cache lines and threads to processors that will result in no false-sharing in our example? In other words, is it possible that the threads assigned to one processor will have their components of `y` in one cache line, and the threads assigned to the other processor will have their components in a different cache line?
f. How many assignments of components to cache lines and threads to processors are there?
g. Of these assignments, how many will result in no false sharing?

5.15. **a.** Modify the matrix-vector multiplication program so that it pads the vector `y` when there's a possibility of false sharing. The padding should be done so that if the threads execute in lock-step, there's no possibility that a single cache line containing an element of `y` will be shared by two or more threads. Suppose, for example, that a cache line stores eight `doubles` and we run the program with four threads. If we allocate storage for at least 48 `doubles` in `y`, then, on each pass through the `for i` loop, there's no possibility that two threads will simultaneously access the same cache line.

b. Modify the matrix-vector multiplication program so that each thread uses private storage for its part of `y` during the `for i` loop. When a thread is done computing its part of `y`, it should copy its private storage into the shared variable.

c. How does the performance of these two alternatives compare to the original program. How do they compare to each other?

5.16. Although `strtok_r` is thread-safe, it has the rather unfortunate property that it gratuitously modifies the input string. Write a tokenizer that is thread-safe and doesn't modify the input string.

5.13 PROGRAMMING ASSIGNMENTS

5.1. Use OpenMP to implement the parallel histogram program discussed in Chapter 2.

5.2. Suppose we toss darts randomly at a square dartboard, whose bullseye is at the origin, and whose sides are 2 feet in length. Suppose also that there's a circle inscribed in the square dartboard. The radius of the circle is 1 foot, and it's area is π square feet. If the points that are hit by the darts are uniformly distributed (and we always hit the square), then the number of darts that hit inside the circle should approximately satisfy the equation

$$\frac{\text{number in circle}}{\text{total number of tosses}} = \frac{\pi}{4},$$

since the ratio of the area of the circle to the area of the square is $\pi/4$.

We can use this formula to estimate the value of π with a random number generator:

```
number_in_circle = 0;
for (toss = 0; toss < number_of_tosses; toss++) {
   x = random double between -1 and 1;
   y = random double between -1 and 1;
   distance_squared = x*x + y*y;
   if (distance_squared <= 1) number_in_circle++;
}
pi_estimate = 4*number_in_circle/((double) number_of_tosses);
```

This is called a "Monte Carlo" method, since it uses randomness (the dart tosses).

Write an OpenMP program that uses a Monte Carlo method to estimate π. Read in the total number of tosses before forking any threads. Use a `reduction` clause to find the total number of darts hitting inside the circle. Print the result after joining all the threads. You may want to use `long long ints` for the number of hits in the circle and the number of tosses, since both may have to be very large to get a reasonable estimate of π.

5.3. Count sort is a simple serial sorting algorithm that can be implemented as follows:

```
void Count_sort(int a[], int n) {
   int i, j, count;
   int* temp = malloc(n*sizeof(int));

   for (i = 0; i < n; i++) {
      count = 0;
      for (j = 0; j < n; j++)
         if (a[j] < a[i])
            count++;
         else if (a[j] == a[i] && j < i)
            count++;
      temp[count] = a[i];
   }

   memcpy(a, temp, n*sizeof(int));
   free(temp);
}  /* Count_sort */
```

The basic idea is that for each element `a[i]` in the list `a`, we count the number of elements in the list that are less than `a[i]`. Then we insert `a[i]` into a temporary list using the subscript determined by the count. There's a slight problem with this approach when the list contains equal elements, since they could get assigned to the same slot in the temporary list. The code deals with this by incrementing the count for equal elements on the basis of the subscripts. If both `a[i] == a[j]` *and* `j <i`, then we count `a[j]` as being "less than" `a[i]`.

After the algorithm has completed, we overwrite the original array with the temporary array using the string library function `memcpy`.

a. If we try to parallelize the `for i` loop (the outer loop), which variables should be private and which should be shared?

b. If we parallelize the `for i` loop using the scoping you specified in the previous part, are there any loop-carried dependences? Explain your answer.

c. Can we parallelize the call to `memcpy`? Can we modify the code so that this part of the function will be parallelizable?

d. Write a C program that includes a parallel implementation of `Count_sort`.

e. How does the performance of your parallelization of `Count_sort` compare to serial `Count_sort`? How does it compare to the serial `qsort` library function?

5.4. Recall that when we solve a large linear system, we often use Gaussian elimination followed by *backward substitution*. Gaussian elimination converts an $n \times n$ linear system into an *upper triangular* linear system by using the "row operations."

- Add a multiple of one row to another row
- Swap two rows
- Multiply one row by a nonzero constant

An upper triangular system has zeroes below the "diagonal" extending from the upper left-hand corner to the lower right-hand corner.

For example, the linear system

$$\begin{aligned} 2x_0 - 3x_1 &= 3 \\ 4x_0 - 5x_1 + x_2 &= 7 \\ 2x_0 - x_1 - 3x_2 &= 5 \end{aligned}$$

can be reduced to the upper triangular form

$$\begin{aligned} 2x_0 - 3x_1 &= 3 \\ x_1 + x_2 &= 1 \\ -5x_2 &= 0 \end{aligned},$$

and this system can be easily solved by first finding x_2 using the last equation, then finding x_1 using the second equation, and finally finding x_0 using the first equation.

We can devise a couple of serial algorithms for back substitution. The "row-oriented" version is

```
for (row = n-1; row >= 0; row--) {
   x[row] = b[row];
   for (col = row+1; col < n; col++)
      x[row] -= A[row][col]*x[col];
   x[row] /= A[row][row];
}
```

Here the "right-hand side" of the system is stored in array `b`, the two-dimensional array of coefficients is stored in array `A`, and the solutions are stored in array `x`. An alternative is the following "column-oriented" algorithm:

```
for (row = 0; row < n; row++)
   x[row] = b[row];

for (col = n-1; col >= 0; col--) {
   x[col] /= A[col][col];
   for (row = 0; row < col; row++)
      x[row] -= A[row][col]*x[col];
}
```

a. Determine whether the outer loop of the row-oriented algorithm can be parallelized.

b. Determine whether the inner loop of the row-oriented algorithm can be parallelized.
c. Determine whether the (second) outer loop of the column-oriented algorithm can be parallelized.
d. Determine whether the inner loop of the column-oriented algorithm can be parallelized.
e. Write one OpenMP program for each of the loops that you determined could be parallelized. You may find the `single` directive useful—when a block of code is being executed in parallel and a sub-block should be executed by only one thread, the sub-block can be modified by a `#pragma omp single` directive. The threads in the executing team will block at the end of the directive until all of the threads have completed it.
f. Modify your parallel loop with a `schedule(runtime)` clause and test the program with various schedules. If your upper triangular system has 10,000 variables, which schedule gives the best performance?

5.5. Use OpenMP to implement a program that does Gaussian elimination. (See the preceding problem.) You can assume that the input system doesn't need any row-swapping.

5.6. Use OpenMP to implement a producer-consumer program in which some of the threads are producers and others are consumers. The producers read text from a collection of files, one per producer. They insert lines of text into a single shared queue. The consumers take the lines of text and tokenize them. Tokens are "words" separated by white space. When a consumer finds a token, it writes it to `stdout`.

CHAPTER

Parallel Program Development

6

In the last three chapters we haven't just learned about parallel APIs, we've also developed a number of small parallel programs, and each of these programs has involved the implementation of a parallel algorithm. In this chapter, we'll look at a couple of larger examples: solving *n*-body problems and solving the traveling salesperson problem. For each problem, we'll start by looking at a serial solution and examining modifications to the serial solution. As we apply Foster's methodology, we'll see that there are some striking similarities between developing shared- and distributed-memory programs. We'll also see that in parallel programming there are problems that we need to solve for which there is no serial analog. We'll see that there are instances in which, as parallel programmers, we'll have to start "from scratch."

6.1 TWO *n*-BODY SOLVERS

In an *n*-body problem, we need to find the positions and velocities of a collection of interacting particles over a period of time. For example, an astrophysicist might want to know the positions and velocities of a collection of stars, while a chemist might want to know the positions and velocities of a collection of molecules or atoms. An *n*-body solver is a program that finds the solution to an *n*-body problem by simulating the behavior of the particles. The input to the problem is the mass, position, and velocity of each particle at the start of the simulation, and the output is typically the position and velocity of each particle at a sequence of user-specified times, or simply the position and velocity of each particle at the end of a user-specified time period.

Let's first develop a serial *n*-body solver. Then we'll try to parallelize it for both shared- and distributed-memory systems.

6.1.1 The problem

For the sake of explicitness, let's write an *n*-body solver that simulates the motions of planets or stars. We'll use Newton's second law of motion and his law of universal gravitation to determine the positions and velocities. Thus, if particle q has position $\mathbf{s}_q(t)$ at time t, and particle k has position $\mathbf{s}_k(t)$, then the force on particle q exerted by

An Introduction to Parallel Programming. DOI: 10.1016/B978-0-12-374260-5.00006-3

particle k is given by

$$\mathbf{f}_{qk}(t) = -\frac{Gm_q m_k}{\left|\mathbf{s}_q(t) - \mathbf{s}_k(t)\right|^3}\left[\mathbf{s}_q(t) - \mathbf{s}_k(t)\right]. \tag{6.1}$$

Here, G is the gravitational constant ($6.673 \times 10^{-11}\text{m}^3/(\text{kg} \cdot \text{s}^2)$), and m_q and m_k are the masses of particles q and k, respectively. Also, the notation $\left|\mathbf{s}_q(t) - \mathbf{s}_k(t)\right|$ represents the distance from particle k to particle q. Note that in general the positions, the velocities, the accelerations, and the forces are vectors, so we're using boldface to represent these variables. We'll use an italic font to represent the other, scalar, variables, such as the time t and the gravitational constant G.

We can use Formula 6.1 to find the total force on any particle by adding the forces due to all the particles. If our n particles are numbered $0, 1, 2, \ldots, n-1$, then the total force on particle q is given by

$$\mathbf{F}_q(t) = \sum_{\substack{k=0 \\ k \neq q}}^{n-1} \mathbf{f}_{qk} = -Gm_q \sum_{\substack{k=0 \\ k \neq q}}^{n-1} \frac{m_k}{\left|\mathbf{s}_q(t) - \mathbf{s}_k(t)\right|^3}\left[\mathbf{s}_q(t) - \mathbf{s}_k(t)\right]. \tag{6.2}$$

Recall that the acceleration of an object is given by the second derivative of its position and that Newton's second law of motion states that the force on an object is given by its mass multiplied by its acceleration, so if the acceleration of particle q is $\mathbf{a}_q(t)$, then $\mathbf{F}_q(t) = m_q\mathbf{a}_q(t) = m_q\mathbf{s}_q''(t)$, where $\mathbf{s}_q''(t)$ is the second derivative of the position $\mathbf{s}_q(t)$. Thus, we can use Formula 6.2 to find the acceleration of particle q:

$$\mathbf{s}_q''(t) = -G\sum_{\substack{j=0 \\ j \neq q}}^{n-1} \frac{m_j}{\left|\mathbf{s}_q(t) - \mathbf{s}_j(t)\right|^3}\left[\mathbf{s}_q(t) - \mathbf{s}_j(t)\right]. \tag{6.3}$$

Thus Newton's laws give us a system of *differential* equations—equations involving derivatives—and our job is to find at each time t of interest the position $\mathbf{s}_q(t)$ and velocity $\mathbf{v}_q(t) = \mathbf{s}_q'(t)$.

We'll suppose that we either want to find the positions and velocities at the times

$$t = 0, \Delta t, 2\Delta t, \ldots, T\Delta t,$$

or, more often, simply the positions and velocities at the final time $T\Delta t$. Here, Δt and T are specified by the user, so the input to the program will be n, the number of particles, Δt, T, and, for each particle, its mass, its initial position, and its initial velocity. In a fully general solver, the positions and velocities would be three-dimensional vectors, but in order to keep things simple, we'll assume that the particles will move in a plane, and we'll use two-dimensional vectors instead.

The output of the program will be the positions and velocities of the n particles at the timesteps $0, \Delta t, 2\Delta t, \ldots$, or just the positions and velocities at $T\Delta t$. To get the output at only the final time, we can add an input option in which the user specifies whether she only wants the final positions and velocities.

6.1.2 Two serial programs

In outline, a serial n-body solver can be based on the following pseudocode:

```
1     Get input data;
2     for each timestep {
3        if (timestep output) Print positions and velocities of
             particles;
4        for each particle q
5           Compute total force on q;
6        for each particle q
7           Compute position and velocity of q;
8     }
9     Print positions and velocities of particles;
```

We can use our formula for the total force on a particle (Formula 6.2) to refine our pseudocode for the computation of the forces in Lines 4–5:

```
for each particle q {
   for each particle k != q {
      x_diff = pos[q][X] - pos[k][X];
      y_diff = pos[q][Y] - pos[k][Y];
      dist = sqrt(x_diff*x_diff + y_diff*y_diff);
      dist_cubed = dist*dist*dist;
      forces[q][X] -= G*masses[q]*masses[k]/dist_cubed * x_diff;
      forces[q][Y] -= G*masses[q]*masses[k]/dist_cubed * y_diff;
   }
}
```

Here, we're assuming that the forces and the positions of the particles are stored as two-dimensional arrays, `forces` and `pos`, respectively. We're also assuming we've defined constants `X = 0` and `Y = 1`. So the x-component of the force on particle q is `forces[q][X]` and the y-component is `forces[q][Y]`. Similarly, the components of the position are `pos[q][X]` and `pos[q][Y]`. (We'll take a closer look at data structures shortly.)

We can use Newton's third law of motion, that is, for every action there is an equal and opposite reaction, to halve the total number of calculations required for the forces. If the force on particle q due to particle k is $\mathbf{f}_{qk}$, then the force on k due to q is $-\mathbf{f}_{qk}$. Using this simplification we can modify our code to compute forces, as shown in Program 6.1. To better understand this pseudocode, imagine the individual forces as a two-dimensional array:

$$\begin{bmatrix} 0 & \mathbf{f}_{01} & \mathbf{f}_{02} & \cdots & \mathbf{f}_{0,n-1} \\ -\mathbf{f}_{01} & 0 & \mathbf{f}_{12} & \cdots & \mathbf{f}_{1,n-1} \\ -\mathbf{f}_{02} & -\mathbf{f}_{12} & 0 & \cdots & \mathbf{f}_{2,n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ -\mathbf{f}_{0,n-1} & -\mathbf{f}_{1,n-1} & -\mathbf{f}_{2,n-1} & \cdots & 0 \end{bmatrix}.$$

(Why are the diagonal entries 0?) Our original solver simply adds all of the entries in row q to get `forces[q]`. In our modified solver, when $q = 0$, the body of the loop

```
for each particle q
   forces[q] = 0;
for each particle q {
   for each particle k > q {
      x_diff = pos[q][X] - pos[k][X];
      y_diff = pos[q][Y] - pos[k][Y];
      dist = sqrt(x_diff*x_diff + y_diff*y_diff);
      dist_cubed = dist*dist*dist;
      force_qk[X] = G*masses[q]*masses[k]/dist_cubed * x_diff;
      force_qk[Y] = G*masses[q]*masses[k]/dist_cubed * y_diff

      forces[q][X] += force_qk[X];
      forces[q][Y] += force_qk[Y];
      forces[k][X] -= force_qk[X];
      forces[k][Y] -= force_qk[Y];
   }
}
```

Program 6.1: A reduced algorithm for computing n-body forces

`for each particle q` will add the entries in row 0 into `forces[0]`. It will also add the kth entry in column 0 into `forces[k]` for $k = 1, 2, \ldots, n-1$. In general, the qth iteration will add the entries to the right of the diagonal (that is, to the right of the 0) in row q into `forces[q]`, and the entries below the diagonal in column q will be added into their respective forces, that is, the kth entry will be added in to `forces[k]`.

Note that in using this modified solver, it's necessary to initialize the `forces` array in a separate loop, since the qth iteration of the loop that calculates the forces will, in general, add the values it computes into `forces[k]` for $k = q+1, q+2, \ldots, n-1$, not just `forces[q]`.

In order to distinguish between the two algorithms, we'll call the n-body solver with the original force calculation, the *basic* algorithm, and the solver with the number of calculations reduced, the *reduced* algorithm.

The position and the velocity remain to be found. We know that the acceleration of particle q is given by

$$\mathbf{a}_q(t) = \mathbf{s}_q''(t) = \mathbf{F}_q(t)/m_q,$$

where $\mathbf{s}_q''(t)$ is the second derivative of the position $\mathbf{s}_q(t)$ and $\mathbf{F}_q(t)$ is the force on particle q. We also know that the velocity $\mathbf{v}_q(t)$ is the first derivative of the position $\mathbf{s}_q'(t)$, so we need to integrate the acceleration to get the velocity, and we need to integrate the velocity to get the position.

We might at first think that we can simply find an antiderivative of the function in Formula 6.3. However, a second look shows us that this approach has problems: the right-hand side contains unknown functions $\mathbf{s}_q$ and $\mathbf{s}_k$—not just the variable t—so we'll instead use a **numerical** method for *estimating* the position and the velocity. This means that rather than trying to find simple closed formulas, we'll approximate

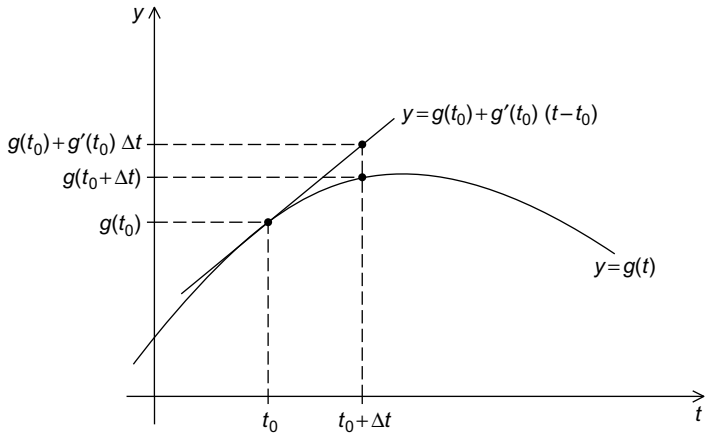


FIGURE 6.1

Using the tangent line to approximate a function

the values of the position and velocity at the times of interest. There are *many* possible choices for numerical methods, but we'll use the simplest one: Euler's method, which is named after the famous Swiss mathematician Leonhard Euler (1707–1783). In Euler's method, we use the tangent line to approximate a function. The basic idea is that if we know the value of a function $g(t_0)$ at time t_0 and we also know its derivative $g'(t_0)$ at time t_0, then we can approximate its value at time $t_0 + \Delta t$ by using the tangent line to the graph of $g(t_0)$. See Figure 6.1 for an example. Now if we know a point $(t_0, g(t_0))$ on a line, and we know the slope of the line $g'(t_0)$, then an equation for the line is given by

$$y = g(t_0) + g'(t_0)(t - t_0).$$

Since we're interested in the time $t = t_0 + \Delta t$, we get

$$g(t + \Delta t) \approx g(t_0) + g'(t_0)(t + \Delta t - t) = g(t_0) + \Delta t g'(t_0).$$

Note that this formula will work even when $g(t)$ and y are vectors: when this is the case, $g'(t)$ is also a vector and the formula just adds a vector to a vector multiplied by a scalar, Δt.

Now we know the value of $\mathbf{s}_q(t)$ and $\mathbf{s}'_q(t)$ at time 0, so we can use the tangent line and our formula for the acceleration to compute $\mathbf{s}_q(\Delta t)$ and $\mathbf{v}_q(\Delta t)$:

$$\mathbf{s}_q(\Delta t) \approx \mathbf{s}_q(0) + \Delta t \mathbf{s}'_q(0) = \mathbf{s}_q(0) + \Delta t \mathbf{v}_q(0),$$

$$\mathbf{v}_q(\Delta t) \approx \mathbf{v}_q(0) + \Delta t \mathbf{v}'_q(0) = \mathbf{v}_q(0) + \Delta t \mathbf{a}_q(0) = \mathbf{v}_q(0) + \Delta t \frac{1}{m_q} \mathbf{F}_q(0).$$

When we try to extend this approach to the computation of $\mathbf{s}_q(2\Delta t)$ and $\mathbf{s}'_q(2\Delta t)$, we see that things are a little bit different, since we don't know the exact value of $\mathbf{s}_q(\Delta t)$

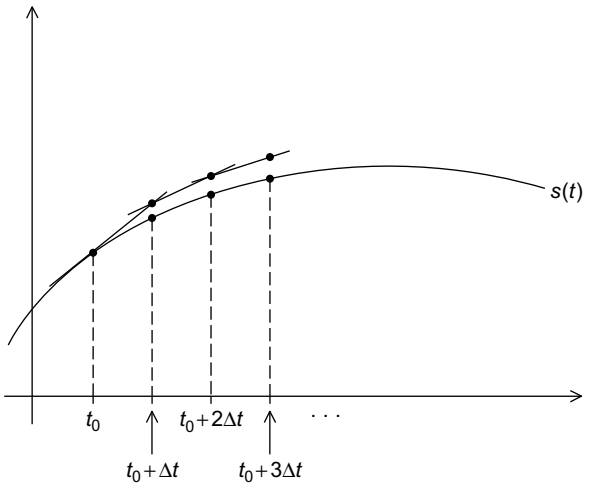


FIGURE 6.2

Euler's method

and $\mathbf{s}'_q(\Delta t)$. However, if our approximations to $\mathbf{s}_q(\Delta t)$ and $\mathbf{s}'_q(\Delta t)$ are good, then we should be able to get a reasonably good approximation to $\mathbf{s}_q(2\Delta t)$ and $\mathbf{s}'_q(2\Delta t)$ using the same idea. This is what Euler's method does (see Figure 6.2).

Now we can complete our pseudocode for the two n-body solvers by adding in the code for computing position and velocity:

```
pos[q][X] += delta_t*vel[q][X];
pos[q][Y] += delta_t*vel[q][Y];
vel[q][X] += delta_t/masses[q]*forces[q][X];
vel[q][Y] += delta_t/masses[q]*forces[q][Y];
```

Here, we're using `pos[q]`, `vel[q]`, and `forces[q]` to store the position, the velocity, and the force, respectively, of particle q.

Before moving on to parallelizing our serial program, let's take a moment to look at data structures. We've been using an array type to store our vectors:

```
#define DIM 2

typedef double vect_t[DIM];
```

A struct is also an option. However, if we're using arrays and we decide to change our program so that it solves three-dimensional problems, in principle, we only need to change the macro `DIM`. If we try to do this with structs, we'll need to rewrite the code that accesses individual components of the vector.

For each particle, we need to know the values of

- its mass,
- its position,

- its velocity,
- its acceleration, and
- the total force acting on it.

Since we're using Newtonian physics, the mass of each particle is constant, but the other values will, in general, change as the program proceeds. If we examine our code, we'll see that once we've computed a new value for one of these variables for a given timestep, we never need the old value again. For example, we don't need to do anything like this

```
new_pos_q = f(old_pos_q);
new_vel_q = g(old_pos_q, new_pos_q);
```

Also, the acceleration is only used to compute the velocity, and its value can be computed in one arithmetic operation from the total force, so we only need to use a local, temporary variable for the acceleration.

For each particle it suffices to store its mass and the current value of its position, velocity, and force. We could store these four variables as a struct and use an array of structs to store the data for all the particles. Of course, there's no reason that all of the variables associated with a particle need to be grouped together in a struct. We can split the data into separate arrays in a variety of different ways. We've chosen to group the mass, position, and velocity into a single struct and store the forces in a separate array. With the forces stored in contiguous memory, we can use a fast function such as `memset` to quickly assign zeroes to all of the elements at the beginning of each iteration:

```
#include <string.h>  /* For memset */
. . .
vect_t* forces = malloc(n*sizeof(vect_t));
. . .
for (step = 1; step <= n_steps; step++) {
   . . .
   /* Assign 0 to each element of the forces array */
   forces = memset(forces, 0, n*sizeof(vect_t);
   for (part = 0; part < n-1; part++)
      Compute_force(part, forces, . . .)
   . . .
}
```

If the force on each particle were a member of a struct, the force members wouldn't occupy contiguous memory in an array of structs, and we'd have to use a relatively slow `for` loop to assign zero to each element.

6.1.3 Parallelizing the n-body solvers

Let's try to apply Foster's methodology to the n-body solver. Since we initially want *lots* of tasks, we can start by making our tasks the computations of the positions, the velocities, and the total forces at each timestep. In the basic algorithm, the algorithm in which the total force on each particle is calculated directly from Formula 6.2, the

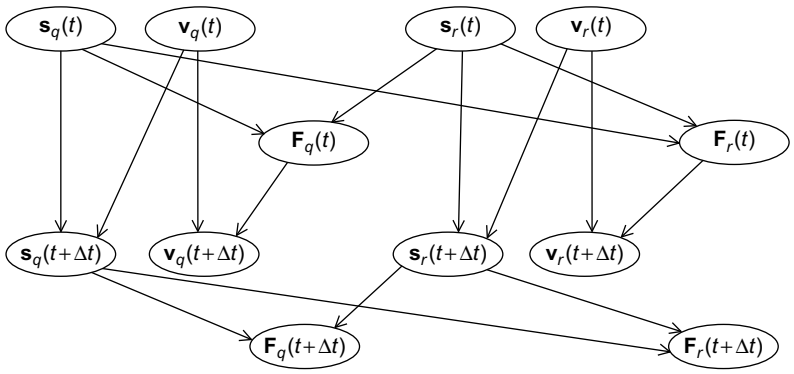


FIGURE 6.3

Communications among tasks in the basic n-body solver

computation of $\mathbf{F}_q(t)$, the total force on particle q at time t, requires the positions of each of the particles $\mathbf{s}_r(t)$, for each r. The computation of $\mathbf{v}_q(t+\Delta t)$ requires the velocity at the previous timestep, $\mathbf{v}_q(t)$, and the force, $\mathbf{F}_q(t)$, at the previous timestep. Finally, the computation of $\mathbf{s}_q(t+\Delta t)$ requires $\mathbf{s}_q(t)$ and $\mathbf{v}_q(t)$. The communications among the tasks can be illustrated as shown in Figure 6.3. The figure makes it clear that most of the communication among the tasks occurs among the tasks associated with an individual particle, so if we agglomerate the computations of $\mathbf{s}_q(t)$, $\mathbf{v}_q(t)$, and $\mathbf{F}_q(t)$, our intertask communication is greatly simplified (see Figure 6.4). Now the tasks correspond to the particles and, in the figure, we've labeled the communications with the data that's being communicated. For example, the arrow from particle q at timestep t to particle r at timestep t is labeled with $\mathbf{s}_q$, the position of particle q.

For the reduced algorithm, the "intra-particle" communications are the same. That is, to compute $\mathbf{s}_q(t+\Delta t)$ we'll need $\mathbf{s}_q(t)$ and $\mathbf{v}_q(t)$, and to compute $\mathbf{v}_q(t+\Delta t)$, we'll need $\mathbf{v}_q(t)$ and $\mathbf{F}_q(t)$. Therefore, once again it makes sense to agglomerate the computations associated with a single particle into a composite task.

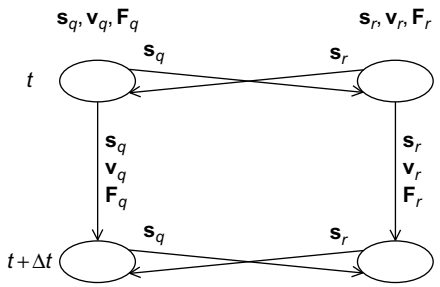


FIGURE 6.4

Communications among agglomerated tasks in the basic n-body solver

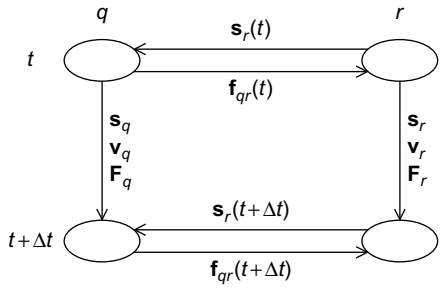


FIGURE 6.5

Communications among agglomerated tasks in the reduced n-body solver ($q < r$)

Recollect that in the reduced algorithm, we make use of the fact that the force $\mathbf{f}_{rq} = -\mathbf{f}_{qr}$. So if $q < r$, then the communication *from* task r *to* task q is the same as in the basic algorithm—in order to compute $\mathbf{F}_q(t)$, task/particle q will need $\mathbf{s}_r(t)$ from task/particle r. However, the communication from task q to task r is no longer $s_q(t)$, it's the force on particle q due to particle r, that is, $\mathbf{f}_{qr}(t)$. See Figure 6.5.

The final stage in Foster's methodology is mapping. If we have n particles and T timesteps, then there will be nT tasks in both the basic and the reduced algorithm. Astrophysical n-body problems typically involve thousands or even millions of particles, so n is likely to be several orders of magnitude greater than the number of available cores. However, T may also be much larger than the number of available cores. So, in principle, we have two "dimensions" to work with when we map tasks to cores. However, if we consider the nature of Euler's method, we'll see that attempting to assign tasks associated with a single particle at different timesteps to different cores won't work very well. Before estimating $\mathbf{s}_q(t + \Delta t)$ and $\mathbf{v}_q(t + \Delta t)$, Euler's method must "know" $\mathbf{s}_q(t)$, $\mathbf{v}_q(t)$, and $\mathbf{a}_q(t)$. Thus, if we assign particle q at time t to core c_0, and we assign particle q at time $t + \Delta t$ to core $c_1 \neq c_0$, then we'll have to communicate $\mathbf{s}_q(t)$, $\mathbf{v}_q(t)$, and $\mathbf{F}_q(t)$ from c_0 to c_1. Of course, if particle q at time t and particle q at time $t + \Delta t$ are mapped to the same core, this communication won't be necessary, so once we've mapped the task consisting of the calculations for particle q at the first timestep to core c_0, we may as well map the subsequent computations for particle q to the same cores, since we can't simultaneously execute the computations for particle q at two different timesteps. Thus, mapping tasks to cores will, in effect, be an assignment of particles to cores.

At first glance, it might seem that any assignment of particles to cores that assigns roughly n/`thread_count` particles to each core will do a good job of balancing the workload among the cores, and for the basic algorithm this is the case. In the basic algorithm the work required to compute the position, velocity, and force is the same for every particle. However, in the reduced algorithm the work required in the forces computation loop is much greater for lower-numbered iterations than the work required for higher-numbered iterations. To see this, recall the pseudocode that computes the total force on particle q in the reduced algorithm:

```
for each particle k > q {
   x_diff = pos[q][X] - pos[k][X];
   y_diff = pos[q][Y] - pos[k][Y];
   dist = sqrt(x_diff*x_diff + y_diff*y_diff);
   dist_cubed = dist*dist*dist;
   force_qk[X] = G*masses[q]*masses[k]/dist_cubed * x_diff;
   force_qk[Y] = G*masses[q]*masses[k]/dist_cubed * y_diff;

   forces[q][X] += force_qk[X];
   forces[q][Y] += force_qk[Y];
   forces[k][X] -= force_qk[X];
   forces[k][Y] -= force_qk[Y];
}
```

Then, for example, when $q = 0$, we'll make $n - 1$ passes through the `for each particle k > q` loop, while when $q = n - 1$, we won't make any passes through the loop. Thus, for the reduced algorithm we would expect that a cyclic partition of the particles would do a better job than a block partitition of evenly distributing the *computation*.

However, in a shared-memory setting, a cyclic partition of the particles among the cores is almost certain to result in a much higher number of cache misses than a block partition, and in a distributed-memory setting, the overhead involved in communicating data that has a cyclic distribution will probably be greater than the overhead involved in communicating data that has a block distribution (see Exercises 6.8 and 6.9).

Therefore with a composite task consisting of all of the computations associated with a single particle throughout the simulation, we conclude the following:

1. A block distribution will give the best performance for the basic n-body solver.
2. For the reduced n-body solver, a cyclic distribution will best distribute the workload in the computation of the forces. However, this improved performance *may* be offset by the cost of reduced cache performance in a shared-memory setting and additional communication overhead in a distributed-memory setting.

In order to make a final determination of the optimal mapping of tasks to cores, we'll need to do some experimentation.

6.1.4 A word about I/O

You may have noticed that our discussion of parallelizing the n-body solver hasn't touched on the issue of I/O, even though I/O can figure prominently in both of our serial algorithms. We've discussed the problem of I/O several times in earlier chapters. Recall that different parallel systems vary widely in their I/O capabilities, and with the very basic I/O that is commonly available it is very difficult to obtain high performance. This basic I/O was designed for use by single-process, single-threaded programs, and when multiple processes or multiple threads attempt to access the I/O buffers, the system makes no attempt to schedule their access. For example, if multiple threads attempt to execute

```
printf("Hello from thread %d of %d\n", my_rank, thread_count);
```

more or less simultaneously, the order in which the output appears will be unpredictable. Even worse, one thread's output may not even appear as a single line. It can happen that the output from one thread appears as multiple segments, and the individual segments are separated by output from other threads.

Thus, as we've noted earlier, except for debug output, we generally assume that one process/thread does all the I/O, and when we're timing program execution, we'll use the option to only print output for the final timestep. Furthermore, we won't include this output in the reported run-times.

Of course, even if we're ignoring the cost of I/O, we can't ignore its existence. We'll briefly discuss its implementation when we discuss the details of our parallel implementations.

6.1.5 Parallelizing the basic solver using OpenMP

How can we use OpenMP to map tasks/particles to cores in the basic version of our n-body solver? Let's take a look at the pseudocode for the serial program:

```
for each timestep {
  if (timestep output) Print positions and velocities of particles;
  for each particle q
     Compute total force on q;
  for each particle q
     Compute position and velocity of q;
}
```

The two inner loops are both iterating over particles. So, in principle, parallelizing the two inner `for` loops will map tasks/particles to cores, and we might try something like this:

```
    for each timestep {
       if (timestep output) Print positions and velocities of
           particles;
#      pragma omp parallel for
       for each particle q
          Compute total force on q;
#      pragma omp parallel for
       for each particle q
          Compute position and velocity of q;
    }
```

We may not like the fact that this code could do a lot of forking and joining of threads, but before dealing with that, let's take a look at the loops themselves: we need to see if there are any race conditions caused by loop-carried dependences.

In the basic version the first loop has the following form:

```
#  pragma omp parallel for
   for each particle q {
      forces[q][X] = forces[q][Y] = 0;
      for each particle k != q {
         x_diff = pos[q][X] - pos[k][X];
         y_diff = pos[q][Y] - pos[k][Y];
```

```
            dist = sqrt(x_diff*x_diff + y_diff*y_diff);
            dist_cubed = dist*dist*dist;
            forces[q][X] -= G*masses[q]*masses[k]/dist_cubed * x_diff;
            forces[q][Y] -= G*masses[q]*masses[k]/dist_cubed * y_diff;
         }
      }
```

Since the iterations of the `for each particle q` loop are partitioned among the threads, only one thread will access `forces[q]` for any q. Different threads do access the same elements of the `pos` array and the `masses` array. However, these arrays are only *read* in the loop. The remaining variables are used for temporary storage in a single iteration of the inner loop, and they can be private. Thus, the parallelization of the first loop in the basic algorithm won't introduce any race conditions.

The second loop has the form:

```
#  pragma omp parallel for
   for each particle q {
      pos[q][X] += delta_t*vel[q][X];
      pos[q][Y] += delta_t*vel[q][Y];
      vel[q][X] += delta_t/masses[q]*forces[q][X];
      vel[q][Y] += delta_t/masses[q]*forces[q][Y];
   }
```

Here, a single thread accesses `pos[q]`, `vel[q]`, `masses[q]`, and `forces[q]` for any particle q, and the scalar variables are only read, so parallelizing this loop also won't introduce any race conditions.

Let's return to the issue of repeated forking and joining of threads. In our pseudocode, we have

```
   for each timestep {
      if (timestep output) Print positions and velocities of
          particles;
#     pragma omp parallel for
      for each particle q
         Compute total force on q;
#     pragma omp parallel for
      for each particle q
         Compute position and velocity of q;
   }
```

We encountered a similar issue when we parallelized odd-even transposition sort (see Section 5.6.2). In that case, we put a `parallel` directive before the outermost loop and used OpenMP `for` directives for the inner loops. Will a similar strategy work here? That is, can we do something like this?

```
#  pragma omp parallel
   for each timestep {
      if (timestep output) Print positions and velocities of
          particles;
#     pragma omp for
      for each particle q
```

```
         Compute total force on q;
#     pragma omp for
      for each particle q
         Compute position and velocity of q;
   }
```

This will have the desired effect on the two `for each particle` loops: the same team of threads will be used in both loops and for every iteration of the outer loop. However, we have a clear problem with the output statement. As it stands now, every thread will print all the positions and velocities, and we only want one thread to do the I/O. However, OpenMP provides the `single` directive for exactly this situation: we have a team of threads executing a block of code, but a part of the code should only be executed by one of the threads. Adding the `single` directive gives us the following pseudocode:

```
#  pragma omp parallel
   for each timestep {
      if (timestep output) {
#        pragma omp single
         Print positions and velocities of particles;
      }
#     pragma omp for
      for each particle q
         Compute total force on q;
#     pragma omp for
      for each particle q
         Compute position and velocity of q;
   }
```

There are still a few issues that we need to address. The most important has to do with possible race conditions introduced in the transition from one statement to another. For example, suppose thread 0 completes the first `for each particle` loop before thread 1, and it then starts updating the positions and velocities of its assigned particles in the second `for each particle` loop. Clearly, this could cause thread 1 to use an updated position in the first `for each particle` loop. However, recall that there is an implicit barrier at the end of each structured block that has been parallelized with a `for` directive. So, if thread 0 finishes the first inner loop before thread 1, it will block until thread 1 (and any other threads) finish the first inner loop, and it won't start the second inner loop until all the threads have finished the first. This will also prevent the possibility that a thread might rush ahead and print positions and velocities before they've all been updated by the second loop.

There's also an implicit barrier after the `single` directive, although in this program the barrier isn't necessary. Since the output statement won't update any memory locations, it's OK for some threads to go ahead and start executing the next iteration before output has been completed. Furthermore, the first inner `for` loop in the next iteration only updates the `forces` array, so it can't cause a thread executing the output statement to print incorrect values, and because of the barrier at the end of the first inner loop, no thread can race ahead and start updating positions and velocities in

the second inner loop before the output has been completed. Thus, we could modify the `single` directive with a `nowait` clause. If the OpenMP implementation supports it, this simply eliminates the implied barrier associated with the `single` directive. It can also be used with `for`, `parallel for`, and `parallel` directives. Note that in this case, addition of the `nowait` clause is unlikely to have much effect on performance, since the two `for each particle` loops have implied barriers that will prevent any one thread from getting more than a few statements ahead of any other.

Finally, we may want to add a schedule clause to each of the `for` directives in order to insure that the iterations have a block partition:

```
#     pragma omp for schedule(static, n/thread_count)
```

6.1.6 Parallelizing the reduced solver using OpenMP

The reduced solver has an additional inner loop: the initialization of the `forces` array to 0. If we try to use the same parallelization for the reduced solver, we should also parallelize this loop with a `for` directive. What happens if we try this? That is, what happens if we try to parallelize the reduced solver with the following pseudocode?

```
#  pragma omp parallel
   for each timestep {
      if (timestep output) {
#        pragma omp single
         Print positions and velocities of particles;
      }
#     pragma omp for
      for each particle q
         forces[q] = 0.0;
#     pragma omp for
      for each particle q
         Compute total force on q;
#     pragma omp for
      for each particle q
         Compute position and velocity of q;
   }
```

Parallelization of the initialization of the forces should be fine, as there's no dependence among the iterations. The updating of the positions and velocities is the same in both the basic and reduced solvers, so if the computation of the forces is OK, then this should also be OK.

How does parallelization affect the correctness of the loop for computing the forces? Recall that in the reduced version, this loop has the following form:

```
#  pragma omp for /* Can be faster than memset */
   for each particle q {
      force_qk[X] = force_qk[Y] = 0;
      for each particle k > q {
```

```
            x_diff = pos[q][X] - pos[k][X];
            y_diff = pos[q][Y] - pos[k][Y];
            dist = sqrt(x_diff*x_diff + y_diff*y_diff);
            dist_cubed = dist*dist*dist;
            force_qk[X] = G*masses[q]*masses[k]/dist_cubed * x_diff;
            force_qk[Y] = G*masses[q]*masses[k]/dist_cubed * y_diff;

            forces[q][X] += force_qk[X];
            forces[q][Y] += force_qk[Y];
            forces[k][X] -= force_qk[X];
            forces[k][Y] -= force_qk[Y];
         }
      }
```

As before, the variables of interest are `pos`, `masses`, and `forces`, since the values in the remaining variables are only used in a single iteration, and hence, can be private. Also, as before, elements of the `pos` and `masses` arrays are only read, not updated. We therefore need to look at the elements of the `forces` array. In this version, unlike the basic version, a thread *may* update elements of the `forces` array other than those corresponding to its assigned particles. For example, suppose we have two threads and four particles and we're using a block partition of the particles. Then the total force on particle 3 is given by

$$\mathbf{F}_3 = -\mathbf{f}_{03} - \mathbf{f}_{13} - \mathbf{f}_{23}.$$

Furthermore, thread 0 will compute $\mathbf{f}_{03}$ and $\mathbf{f}_{13}$, while thread 1 will compute $\mathbf{f}_{23}$. Thus, the updates to `forces[3]` *do* create a race condition. In general, then, the updates to the elements of the `forces` array introduce race conditions into the code.

A seemingly obvious solution to this problem is to use a `critical` directive to limit access to the elements of the `forces` array. There are at least a couple of ways to do this. The simplest is to put a `critical` directive before all the updates to `forces`

```
#  pragma omp critical
   {
      forces[q][X] += force_qk[X];
      forces[q][Y] += force_qk[Y];
      forces[k][X] -= force_qk[X];
      forces[k][Y] -= force_qk[Y];
   }
```

However, with this approach access to the elements of the `forces` array will be effectively serialized. Only one element of `forces` can be updated at a time, and contention for access to the critical section is actually likely to seriously degrade the performance of the program. See Exercise 6.3.

An alternative would be to have one critical section for each particle. However, as we've seen, OpenMP doesn't readily support varying numbers of critical sections, so we would need to use one lock for each particle instead and our updates would

look something like this:

```
omp_set_lock(&locks[q]);
forces[q][X] += force_qk[X];
forces[q][Y] += force_qk[Y];
omp_unset_lock(&locks[q]);

omp_set_lock(&locks[k]);
forces[k][X] -= force_qk[X];
forces[k][Y] -= force_qk[Y];
omp_unset_lock(&locks[k]);
```

This assumes that the master thread will create a shared array of locks, one for each particle, and when we update an element of the `forces` array, we first set the lock corresponding to that particle. Although this approach performs much better than the single critical section, it still isn't competitive with the serial code. See Exercise 6.4.

Another possible solution is to carry out the computation of the forces in two phases. In the first phase, each thread carries out exactly the same calculations it carried out in the erroneous parallelization. However, now the calculations are stored in its *own* array of forces. Then, in the second phase, the thread that has been assigned particle q will add the contributions that have been computed by the different threads. In our example above, thread 0 would compute $-\mathbf{f}_{03} - \mathbf{f}_{13}$, while thread 1 would compute $-\mathbf{f}_{23}$. After each thread was done computing its contributions to the forces, thread 1, which has been assigned particle 3, would find the total force on particle 3 by adding these two values.

Let's look at a slightly larger example. Suppose we have three threads and six particles. If we're using a block partition of the particles, then the computations in the first phase are shown in Table 6.1. The last three columns of the table show each thread's contribution to the computation of the total forces. In phase 2 of the computation, the thread specified in the first column of the table will add the contents of each of its assigned rows—that is, each of its assigned particles.

Note that there's nothing special about using a block partition of the particles. Table 6.2 shows the same computations if we use a cyclic partition of the particles.

Table 6.1 First-Phase Computations for a Reduced Algorithm with Block Partition

		Thread		
Thread	**Particle**	**0**	**1**	**2**
0	0	$\mathbf{f}_{01}+\mathbf{f}_{02}+\mathbf{f}_{03}+\mathbf{f}_{04}+\mathbf{f}_{05}$	0	0
	1	$-\mathbf{f}_{01}+\mathbf{f}_{12}+\mathbf{f}_{13}+\mathbf{f}_{14}+\mathbf{f}_{15}$	0	0
1	2	$-\mathbf{f}_{02}-\mathbf{f}_{12}$	$\mathbf{f}_{23}+\mathbf{f}_{24}+\mathbf{f}_{25}$	0
	3	$-\mathbf{f}_{03}-\mathbf{f}_{13}$	$-\mathbf{f}_{23}+\mathbf{f}_{34}+\mathbf{f}_{35}$	0
2	4	$-\mathbf{f}_{04}-\mathbf{f}_{14}$	$-\mathbf{f}_{24}-\mathbf{f}_{34}$	$\mathbf{f}_{45}$
	5	$-\mathbf{f}_{05}-\mathbf{f}_{15}$	$-\mathbf{f}_{25}-\mathbf{f}_{35}$	$-\mathbf{f}_{45}$

Table 6.2 First-Phase Computations for a Reduced Algorithm with Cyclic Partition

Thread	Particle	Thread 0	Thread 1	Thread 2
0	0	$\mathbf{f}_{01}+\mathbf{f}_{02}+\mathbf{f}_{03}+\mathbf{f}_{04}+\mathbf{f}_{05}$	0	0
1	1	$-\mathbf{f}_{01}$	$\mathbf{f}_{12}+\mathbf{f}_{13}+\mathbf{f}_{14}+\mathbf{f}_{15}$	0
2	2	$-\mathbf{f}_{02}$	$-\mathbf{f}_{12}$	$\mathbf{f}_{23}+\mathbf{f}_{24}+\mathbf{f}_{25}$
0	3	$-\mathbf{f}_{03}+\mathbf{f}_{34}+\mathbf{f}_{35}$	$-\mathbf{f}_{13}$	$-\mathbf{f}_{23}$
1	4	$-\mathbf{f}_{04}-\mathbf{f}_{34}$	$-\mathbf{f}_{14}+\mathbf{f}_{45}$	$-\mathbf{f}_{24}$
2	5	$-\mathbf{f}_{05}-\mathbf{f}_{35}$	$-\mathbf{f}_{15}-\mathbf{f}_{45}$	$-\mathbf{f}_{25}$

Note that if we compare this table with the table that shows the block partition, it's clear that the cyclic partition does a better job of balancing the load.

To implement this, during the first phase our revised algorithm proceeds as before, except that each thread adds the forces it computes into its own subarray of `loc_forces`:

```
#  pragma omp for
   for each particle q {
      force_qk[X] = force_qk[Y] = 0;
      for each particle k > q {
         x_diff = pos[q][X] - pos[k][X];
         y_diff = pos[q][Y] - pos[k][Y];
         dist = sqrt(x_diff*x_diff + y_diff*y_diff);
         dist_cubed = dist*dist*dist;
         force_qk[X] = G*masses[q]*masses[k]/dist_cubed * x_diff;
         force_qk[Y] = G*masses[q]*masses[k]/dist_cubed * y_diff;

         loc_forces[my_rank][q][X] += force_qk[X];
         loc_forces[my_rank][q][Y] += force_qk[Y];
         loc_forces[my_rank][k][X] -= force_qk[X];
         loc_forces[my_rank][k][Y] -= force_qk[Y];
      }
   }
```

During the second phase, each thread adds the forces computed by all the threads for its assigned particles:

```
#  pragma omp for
   for (q = 0; q < n; q++) {
      forces[q][X] = forces[q][Y] = 0;
      for (thread = 0; thread < thread_count; thread++) {
         forces[q][X] += loc_forces[thread][q][X];
         forces[q][Y] += loc_forces[thread][q][Y];
      }
   }
```

Before moving on, we should make sure that we haven't inadvertently introduced any new race conditions. During the first phase, since each thread writes to its own

subarray, there isn't a race condition in the updates to `loc_forces`. Also, during the second phase, only the "owner" of thread q writes to `forces[q]`, so there are no race conditions in the second phase. Finally, since there is an implied barrier after each of the parallelized `for` loops, we don't need to worry that some thread is going to race ahead and make use of a variable that hasn't been properly initialized, or that some slow thread is going to make use of a variable that has had its value changed by another thread.

6.1.7 Evaluating the OpenMP codes

Before we can compare the basic and the reduced codes, we need to decide how to schedule the parallelized `for` loops. For the basic code, we've seen that any schedule that divides the iterations equally among the threads should do a good job of balancing the computational load. (As usual, we're assuming no more than one thread/core.) We also observed that a block partitioning of the iterations would result in fewer cache misses than a cyclic partition. Thus, we would expect that a block schedule would be the best option for the basic version.

In the reduced code, the amount of work done in the first phase of the computation of the forces decreases as the `for` loop proceeds. We've seen that a cyclic schedule should do a better job of assigning more or less equal amounts of work to each thread. In the remaining parallel `for` loops—the initialization of the `loc_forces` array, the second phase of the computation of the forces, and the updating of the positions and velocities—the work required is roughly the same for all the iterations. Therefore, *taken out of context* each of these loops will probably perform best with a block schedule. However, the schedule of one loop can affect the performance of another (see Exercise 6.10), so it may be that choosing a cyclic schedule for one loop and block schedules for the others will degrade performance.

With these choices, Table 6.3 shows the performance of the n-body solvers when they're run on one of our systems with no I/O. The solver used 400 particles for 1000 timesteps. The column labeled "Default Sched" gives times for the OpenMP reduced solver when all of the inner loops use the default schedule, which, on our system, is a block schedule. The column labeled "Forces Cyclic" gives times when the first phase of the forces computation uses a cyclic schedule and the other inner loops use the default schedule. The last column, labeled "All Cyclic," gives times when all of

Table 6.3 Run-Times of the n-Body Solvers Parallelized with OpenMP (times are in seconds)

Threads	Basic	Reduced Default Sched	Reduced Forces Cyclic	Reduced All Cyclic
1	7.71	3.90	3.90	3.90
2	3.87	2.94	1.98	2.01
4	1.95	1.73	1.01	1.08
8	0.99	0.95	0.54	0.61

the inner loops use a cyclic schedule. The run-times of the serial solvers differ from those of the single-threaded solvers by less than 1%, so we've omitted them from the table.

Notice that with more than one thread the reduced solver, using all default schedules, takes anywhere from 50 to 75% longer than the reduced solver with the cyclic forces computation. Using the cyclic schedule is clearly superior to the default schedule in this case, and any loss in time resulting from cache issues is more than made up for by the improved load balance for the computations.

For only two threads there is very little difference between the performance of the reduced solver with only the first forces loop cyclic and the reduced solver with all loops cyclic. However, as we increase the number of threads, the performance of the reduced solver that uses a cyclic schedule for all of the loops does start to degrade. In this particular case, when there are more threads, it appears that the overhead involved in changing distributions is less than the overhead incurred from false sharing.

Finally, notice that the basic solver takes about twice as long as the reduced solver with the cyclic scheduling of the forces computation. So if the extra memory is available, the reduced solver is clearly superior. However, the reduced solver increases the memory requirement for the storage of the forces by a factor of `thread_count`, so for very large numbers of particles, it may be impossible to use the reduced solver.

6.1.8 Parallelizing the solvers using pthreads

Parallelizing the two n-body solvers using Pthreads is very similar to parallelizing them using OpenMP. The differences are only in implementation details, so rather than repeating the discussion, we will point out some of the principal differences between the Pthreads and the OpenMP implementations. We will also note some of the more important similarities.

- By default local variables in Pthreads are private, so all shared variables are global in the Pthreads version.
- The principal data structures in the Pthreads version are identical to those in the OpenMP version: vectors are two-dimensional arrays of `doubles`, and the mass, position, and velocity of a single particle are stored in a struct. The forces are stored in an array of vectors.
- Startup for Pthreads is basically the same as the startup for OpenMP: the main thread gets the command-line arguments, and allocates and initializes the principal data structures.
- The main difference between the Pthreads and the OpenMP implementations is in the details of parallelizing the inner loops. Since Pthreads has nothing analogous to a `parallel for` directive, we must explicitly determine which values of the loop variables correspond to each thread's calculations. To facilitate this, we've written a function `Loop_schedule`, which determines
  - the initial value of the loop variable,
  - the final value of the loop variable, and
  - the increment for the loop variable.

The input to the function is
- the calling thread's rank,
- the number of threads,
- the total number of iterations, and
- an argument indicating whether the partitioning should be block or cyclic.

- Another difference between the Pthreads and the OpenMP versions has to do with barriers. Recall that the end of a `parallel for` directive in OpenMP has an implied barrier. As we've seen, this is important. For example, we don't want a thread to start updating its positions until all the forces have been calculated, because it could use an out-of-date force and another thread could use an out-of-date position. If we simply partition the loop iterations among the threads in the Pthreads version, there won't be a barrier at the end of an inner `for` loop and we'll have a race condition. Thus, we need to add explicit barriers after the inner loops when a race condition can arise. The Pthreads standard includes a barrier. However, some systems don't implement it, so we've defined a function that uses a Pthreads condition variable to implement a barrier. See Subsection 4.8.3 for details.

6.1.9 Parallelizing the basic solver using MPI

With our composite tasks corresponding to the individual particles, it's fairly straightforward to parallelize the basic algorithm using MPI. The only communication among the tasks occurs when we're computing the forces, and, in order to compute the forces, each task/particle needs the position and mass of every other particle. `MPI_Allgather` is expressly designed for this situation, since it collects on each process the same information from every other process. We've already noted that a block distribution will probably have the best performance, so we should use a block mapping of the particles to the processes.

In the shared-memory implementations, we collected most of the data associated with a single particle (mass, position, and velocity) into a single struct. However, if we use this data structure in the MPI implementation, we'll need to use a derived datatype in the call to `MPI_Allgather`, and communications with derived datatypes tend to be slower than communications with basic MPI types. Thus, it will make more sense to use individual arrays for the masses, positions, and velocities. We'll also need an array for storing the positions of all the particles. If each process has sufficient memory, then each of these can be a separate array. In fact, if memory isn't a problem, each process can store the entire array of masses, since these will never be updated and their values only need to be communicated during the initial setup.

On the other hand, if memory is short, there is an "in-place" option that can be used with some MPI collective communications. For our situation, suppose that the array `pos` can store the positions of all n particles. Further suppose that `vect_mpi_t` is an MPI datatype that stores two contiguous `doubles`. Also suppose that n is evenly divisible by `comm_sz` and `loc_n = n/comm_sz`. Then, if we store the local positions in a separate array, `loc_pos`, we can use the following call to collect all of the positions

on each process:

```
MPI_Allgather(loc_pos, loc_n, vect_mpi_t,
      pos, loc_n, vect_mpi_t, comm);
```

If we can't afford the extra storage for `loc_pos`, then we can have each process q store its local positions in the qth block of `pos`. That is, the local positions of each process should be stored in the appropriate block of each process' `pos` array:

```
Process 0:  pos[0], pos[1], . . . , pos[loc_n−1]
Process 1:  pos[loc_n], pos[loc_n+1], . . . , pos[loc_n + loc_n−1]
   . . .
Process q:  pos[q*loc_n], pos[q*loc_n+1], . . . , pos[q*loc_n +
     loc_n−1]
   . . .
```

With the `pos` array initialized this way on each process, we can use the following call to `MPI_Allgather`:

```
MPI_Allgather(MPI_IN_PLACE, loc_n, vect_mpi_t,
  pos, loc_n, vect_mpi_t, comm);
```

In this call, the first `loc_n` and `vect_mpi_t` arguments are ignored. However, it's not a bad idea to use arguments whose values correspond to the values that will be used, just to increase the readability of the program.

In the program we've written, we made the following choices with respect to the data structures:

- Each process stores the entire global array of particle masses.
- Each process only uses a single n-element array for the positions.
- Each process uses a pointer `loc_pos` that refers to the start of its block of `pos`. Thus, on process, 0 `local_pos = pos`, on process 1 `local_pos = pos + loc_n`, and, so on.

With these choices, we can implement the basic algorithm with the pseudocode shown in Program 6.2. Process 0 will read and broadcast the command line arguments. It will also read the input and print the results. In Line 1, it will need to distribute the input data. Therefore, `Get input data` might be implemented as follows:

```
if (my_rank == 0) {
   for each particle
      Read masses[particle], pos[particle], vel[particle];
}
MPI_Bcast(masses, n, MPI_DOUBLE, 0, comm);
MPI_Bcast(pos, n, vect_mpi_t, 0, comm);
MPI_Scatter(vel, loc_n, vect_mpi_t, loc_vel, loc_n, vect_mpi_t, 0,
    comm);
```

So process 0 reads all the initial conditions into three n-element arrays. Since we're storing all the masses on each process, we broadcast `masses`. Also, since each process

```
 1  Get input data;
 2  for each timestep {
 3     if (timestep output)
 4        Print positions and velocities of particles;
 5     for each local particle loc_q
 6        Compute total force on loc_q;
 7     for each local particle loc_q
 8        Compute position and velocity of loc_q;
 9     Allgather local positions into global pos array;
10  }
11  Print positions and velocities of particles;
```

Program 6.2: Pseudocode for the MPI version of the basic *n*-body solver

will need the global array of positions for the first computation of forces in the main `for` loop, we just broadcast `pos`. However, velocities are only used locally for the updates to positions and velocities, so we scatter `vel`.

Notice that we gather the updated positions in Line 9 at the end of the body of the outer `for` loop of Program 6.2. This insures that the positions will be available for output in both Line 4 and Line 11. If we're printing the results for each timestep, this placement allows us to eliminate an expensive collective communication call: if we simply gathered the positions onto process 0 before output, we'd have to call `MPI_Allgather` before the computation of the forces. With this organization of the body of the outer `for` loop, we can implement the output with the following pseudocode:

```
Gather velocities onto process 0;
if (my_rank == 0) {
   Print timestep;
   for each particle
      Print pos[particle] and vel[particle]
}
```

6.1.10 Parallelizing the reduced solver using MPI

The "obvious" implementation of the reduced algorithm is likely to be extremely complicated. Before computing the forces, each process will need to gather a subset of the positions, and after the computation of the forces, each process will need to scatter some of the individual forces it has computed and add the forces it receives. Figure 6.6 shows the communications that would take place if we had three processes, six particles, and used a block partitioning of the particles among the processes. Not suprisingly, the communications are even more complex when we use a cyclic distribution (see Exercise 6.13). Certainly it would be possible to implement these communications. However, unless the implementation were *very* carefully done, it would probably be *very* slow.

Fortunately, there's a much simpler alternative that uses a communication structure that is sometimes called a **ring pass**. In a ring pass, we imagine the processes

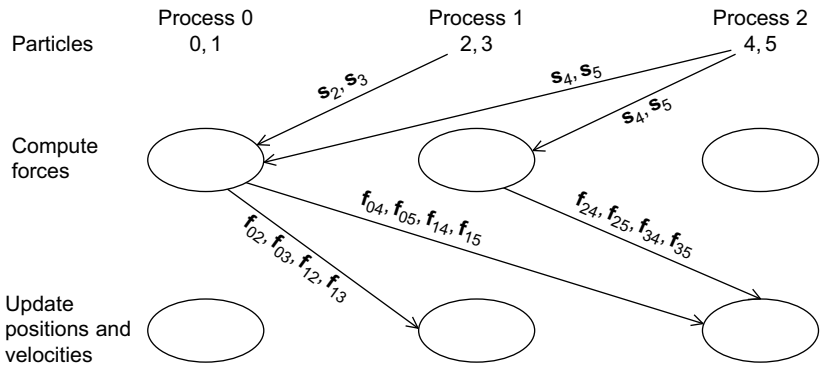


FIGURE 6.6

Communication in a possible MPI implementation of the reduced n-body solver

as being interconnected in a ring (see Figure 6.7). Process 0 communicates directly with processes 1 and `comm_sz` $-$ 1, process 1 communicates with processes 0 and 2, and so on. The communication in a ring pass takes place in phases, and during each phase each process sends data to its "lower-ranked" neighbor, and receives data from its "higher-ranked" neighbor. Thus, 0 will send to `comm_sz` $-$ 1 and receive from 1. 1 will send to 0 and receive from 2, and so on. In general, process q will send to process $(q - 1 + \texttt{comm\_sz})\,\%\,\texttt{comm\_sz}$ and receive from process $(q + 1)\,\%\,\texttt{comm\_sz}$.

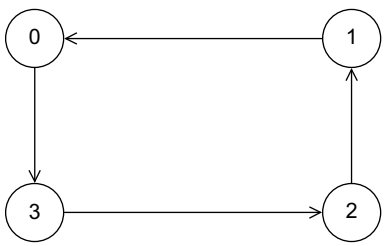


FIGURE 6.7

A ring of processes

By repeatedly sending and receiving data using this ring structure, we can arrange that each process has access to the positions of all the particles. During the first phase, each process will send the positions of its assigned particles to its "lower-ranked" neighbor and receive the positions of the particles assigned to its higher-ranked neighbor. During the next phase, each process will forward the positions it received in the first phase. This process continues through `comm_sz` $-$ 1 phases until each process has received the positions of all of the particles. Figure 6.8 shows the three phases if there are four processes and eight particles that have been cyclically distributed.

Of course, the virtue of the reduced algorithm is that we don't need to compute all of the inter-particle forces since $\mathbf{f}_{kq} = -\mathbf{f}_{qk}$, for every pair of particles q and k. To see

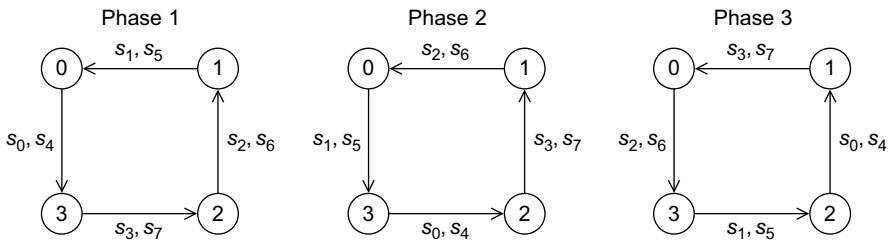


FIGURE 6.8

Ring pass of positions

how to exploit this, first observe that using the reduced algorithm, the interparticle forces can be divided into those that are *added* into and those that are subtracted from the total forces on the particle. For example, if we have six particles, then the reduced algorithm will compute the force on particle 3 as

$$\mathbf{F}_3 = -\mathbf{f}_{03} - \mathbf{f}_{13} - \mathbf{f}_{23} + \mathbf{f}_{34} + \mathbf{f}_{35}.$$

The key to understanding the ring pass computation of the forces is to observe that the interparticle forces that are *subtracted* are computed by another task/particle, while the forces that are *added* are computed by the owning task/particle. Thus, the computations of the interparticle forces on particle 3 are assigned as follows:

Force	$\mathbf{f}_{03}$	$\mathbf{f}_{13}$	$\mathbf{f}_{23}$	$\mathbf{f}_{34}$	$\mathbf{f}_{35}$
Task/Particle	0	1	2	3	3

So, suppose that for our ring pass, instead of simply passing `loc_n` = n/`comm_sz` positions, we also pass `loc_n` forces. Then in each phase, a process can

1. compute interparticle forces resulting from interaction between its assigned particles and the particles whose positions it has received, and
2. once an interparticle force has been computed, the process can add the force into a local array of forces corresponding to its particles, *and* it can subtract the interparticle force from the received array of forces.

See, for example, [15, 34] for further details and alternatives.

Let's take a look at how the computation would proceed when we have four particles, two processes, and we're using a cyclic distribution of the particles among the processes (see Table 6.4). We're calling the arrays that store the local positions and local forces `loc_pos` and `loc_forces`, respectively. These are not communicated among the processes. The arrays that are communicated among the processes are `tmp_pos` and `tmp_forces`.

Before the ring pass can begin, both arrays storing positions are initialized with the positions of the local particles, and the arrays storing the forces are set to 0. Before the ring pass begins, each process computes those forces that are due to interaction

Table 6.4 Computation of Forces in Ring Pass

Time	Variable	Process 0	Process 1
Start	loc_pos	$\mathbf{s}_0, \mathbf{s}_2$	$\mathbf{s}_1, \mathbf{s}_3$
	loc_forces	0,0	0,0
	tmp_pos	$\mathbf{s}_0, \mathbf{s}_2$	$\mathbf{s}_1, \mathbf{s}_3$
	tmp_forces	0,0	0,0
After Comp of Forces	loc_pos	$\mathbf{s}_0, \mathbf{s}_2$	$\mathbf{s}_1, \mathbf{s}_3$
	loc_forces	$\mathbf{f}_{02}, 0$	$\mathbf{f}_{13}, 0$
	tmp_pos	$\mathbf{s}_0, \mathbf{s}_2$	$\mathbf{s}_1, \mathbf{s}_3$
	tmp_forces	$0, -\mathbf{f}_{02}$	$0, -\mathbf{f}_{13}$
After First Comm	loc_pos	$\mathbf{s}_0, \mathbf{s}_2$	$\mathbf{s}_1, \mathbf{s}_3$
	loc_forces	$\mathbf{f}_{02}, 0$	$\mathbf{f}_{13}, 0$
	tmp_pos	$\mathbf{s}_1, \mathbf{s}_3$	$\mathbf{s}_0, \mathbf{s}_2$
	tmp_forces	$0, -\mathbf{f}_{13}$	$0, -\mathbf{f}_{02}$
After Comp of Forces	loc_pos	$\mathbf{s}_0, \mathbf{s}_2$	$\mathbf{s}_1, \mathbf{s}_3$
	loc_forces	$\mathbf{f}_{01} + \mathbf{f}_{02} + \mathbf{f}_{03}, \mathbf{f}_{23}$	$\mathbf{f}_{12} + \mathbf{f}_{13}, 0$
	tmp_pos	$\mathbf{s}_1, \mathbf{s}_3$	$\mathbf{s}_0, \mathbf{s}_2$
	tmp_forces	$-\mathbf{f}_{01}, -\mathbf{f}_{03} - \mathbf{f}_{13} - \mathbf{f}_{23}$	$0, -\mathbf{f}_{02} - \mathbf{f}_{12}$
After Second Comm	loc_pos	$\mathbf{s}_0, \mathbf{s}_2$	$\mathbf{s}_1, \mathbf{s}_3$
	loc_forces	$\mathbf{f}_{01} + \mathbf{f}_{02} + \mathbf{f}_{03}, \mathbf{f}_{23}$	$\mathbf{f}_{12} + \mathbf{f}_{13}, 0$
	tmp_pos	$\mathbf{s}_0, \mathbf{s}_2$	$\mathbf{s}_1, \mathbf{s}_3$
	tmp_forces	$0, -\mathbf{f}_{02} - \mathbf{f}_{12}$	$-\mathbf{f}_{01}, -\mathbf{f}_{03} - \mathbf{f}_{13} - \mathbf{f}_{23}$
After Comp of Forces	loc_pos	$\mathbf{s}_0, \mathbf{s}_2$	$\mathbf{s}_1, \mathbf{s}_3$
	loc_forces	$\mathbf{f}_{01} + \mathbf{f}_{02} + \mathbf{f}_{03}, -\mathbf{f}_{02} - \mathbf{f}_{12} + \mathbf{f}_{23}$	$-\mathbf{f}_{01} + \mathbf{f}_{12} + \mathbf{f}_{13}, -\mathbf{f}_{03} - \mathbf{f}_{13} - \mathbf{f}_{23}$
	tmp_pos	$\mathbf{s}_0, \mathbf{s}_2$	$\mathbf{s}_1, \mathbf{s}_3$
	tmp_forces	$0, -\mathbf{f}_{02} - \mathbf{f}_{12}$	$-\mathbf{f}_{01}, -\mathbf{f}_{03} - \mathbf{f}_{13} - \mathbf{f}_{23}$

among its assigned particles. Process 0 computes $\mathbf{f}_{02}$ and process 1 computes $\mathbf{f}_{13}$. These values are added into the appropriate locations in `loc_forces` and subtracted from the appropriate locations in `tmp_forces`.

Now, the two processes exchange `tmp_pos` and `tmp_forces` and compute the forces due to interaction among their local particles and the received particles. In the reduced algorithm, the lower ranked task/particle carries out the computation. Process 0 computes $\mathbf{f}_{01}, \mathbf{f}_{03}$, and $\mathbf{f}_{23}$, while process 1 computes $\mathbf{f}_{12}$. As before, the newly computed forces are added into the appropriate locations in `loc_forces` and subtracted from the appropriate locations in `tmp_forces`.

To complete the algorithm, we need to exchange the `tmp` arrays one final time.[1] Once each process has received the updated `tmp_forces`, it can carry out a simple vector sum

```
loc_forces += tmp_forces
```

to complete the algorithm.

[1] Actually, we only need to exchange `tmp_forces` for the final communication.

```
1  source = (my_rank + 1) % comm_sz;
2  dest = (my_rank - 1 + comm_sz) % comm_sz;
3  Copy loc_pos into tmp_pos;
4  loc_forces = tmp_forces = 0;
5
6  Compute forces due to interactions among local particles;
7  for (phase = 1; phase < comm_sz; phase++) {
8     Send current tmp_pos and tmp_forces to dest;
9     Receive new tmp_pos and tmp_forces from source;
10    /* Owner of the positions and forces we're receiving */
11    owner = (my_rank + phase) % comm_sz;
12    Compute forces due to interactions among my particles
13       and owner's particles;
14 }
15 Send current tmp_pos and tmp_forces to dest;
16 Receive new tmp_pos and tmp_forces from source;
```

Program 6.3: Pseudocode for the MPI implementation of the reduced n-body solver

Thus, we can implement the computation of the forces in the reduced algorithm using a ring pass with the pseudocode shown in Program 6.3. Recall that using `MPI_Send` and `MPI_Recv` for the send-receive pairs in Lines 8 and 9 and 15 and 16 is *unsafe* in MPI parlance, since they can hang if the system doesn't provide sufficient buffering. In this setting, recall that MPI provides `MPI_Sendrecv` and `MPI_Sendrecv_replace`. Since we're using the same memory for both the outgoing and the incoming data, we can use `MPI_Sendrecv_replace`.

Also recall that the time it takes to start up a message is substantial. We can probably reduce the cost of the communication by using a single array to store both `tmp_pos` and `tmp_forces`. For example, we could allocate storage for an array `tmp_data` that can store $2 \times$ `loc_n` objects with type `vect_t` and use the first `loc_n` for `tmp_pos` and the last `loc_n` for `tmp_forces`. We can continue to use `tmp_pos` and `tmp_forces` by making these pointers to `tmp_data[0]` and `tmp_data[loc_n]`, respectively.

The principal difficulty in implementing the actual computation of the forces in Lines 12 and 13 lies in determining whether the current process should compute the force resulting from the interaction of a particle q assigned to it and a particle r whose position it has received. If we recall the reduced algorithm (Program 6.1), we see that task/particle q is responsible for computing $\mathbf{f}_{qr}$ if and only if $q < r$. However, the arrays `loc_pos` and `tmp_pos` (or a larger array containing `tmp_pos` and `tmp_forces`) use *local* subscripts, not global subscripts. That is, when we access an element of (say) `loc_pos`, the subscript we use will lie in the range $0, 1, \ldots,$ `loc_n` $- 1$, not $0, 1, \ldots, n-1$; so, if we try to implement the force interaction with the following pseudocode, we'll run into (at least) a couple of problems:

```
for (loc_part1 = 0; loc_part1 < loc_n-1; loc_part1++)
   for (loc_part2 = loc_part1+1; loc_part2 < loc_n; loc_part2++)
```

```
Compute_force(loc_pos[loc_part1], masses[loc_part1],
      tmp_pos[loc_part2], masses[loc_part2],
      loc_forces[loc_part1], tmp_forces[loc_part2]);
```

The first, and most obvious, is that `masses` is a global array and we're using local subscripts to access its elements. The second is that the relative sizes of `loc_part1` and `loc_part2` don't tell us whether we should compute the force due to their interaction. We need to use global subscripts to determine this. For example, if we have four particles and two processes, and the preceding code is being run by process 0, then when `loc_part1 = 0`, the inner loop will skip `loc_part2 = 0` and start with `loc_part2 = 1`; however, if we're using a cyclic distribution, `loc_part1 = 0` corresponds to global particle 0 and `loc_part2 = 0` corresponds to global particle 1, and we *should* compute the force resulting from interaction between these two particles.

Clearly, the problem is that we shouldn't be using local particle indexes, but rather we should be using *global* particle indexes. Thus, using a cyclic distribution of the particles, we could modify our code so that the loops also iterate through global particle indexes:

```
for (loc_part1 = 0, glb_part1 = my_rank;
     loc_part1 < loc_n-1;
     loc_part1++, glb_part1 += comm_sz)
  for (glb_part2 = First_index(glb_part1, my_rank, owner, comm_sz),
       loc_part2 = Global_to_local(glb_part2, owner, loc_n);
       loc_part2 < loc_n;
       loc_part2++, glb_part2 += comm_sz)
       Compute_force(loc_pos[loc_part1], masses[glb_part1],
             tmp_pos[loc_part2], masses[glb_part2],
             loc_forces[loc_part1], tmp_forces[loc_part2]);
```

The function `First_index` should determine a global index `glb_part2` with the following properties:

1. The particle `glb_part2` is assigned to the process with rank `owner`.
2. `glb_part1 < glb_part2 < glb_part1 + comm_sz`.

The function `Global_to_local` should convert a global particle index into a local particle index, and the function `Compute_force` should compute the force resulting from the interaction of two particles. We already know how to implement `Compute_force`. See Exercises 6.15 and 6.16 for the other two functions.

6.1.11 Performance of the MPI solvers

Table 6.5 shows the run-times of the two n-body solvers when they're run with 800 particles for 1000 timesteps on an Infiniband-connected cluster. All the timings were taken with one process per cluster node. The run-times of the serial solvers differed from the single-process MPI solvers by less than 1%, so we haven't included them.

Clearly, the performance of the reduced solver is much superior to the performance of the basic solver, although the basic solver achieves higher efficiencies.

Table 6.5 Performance of the MPI *n*-Body Solvers (times in seconds)

Processes	Basic	Reduced
1	17.30	8.68
2	8.65	4.45
4	4.35	2.30
8	2.20	1.26
16	1.13	0.78

Table 6.6 Run-Times for OpenMP and MPI *n*-Body Solvers (times in seconds)

Processes/ Threads	OpenMP		MPI	
	Basic	Reduced	Basic	Reduced
1	15.13	8.77	17.30	8.68
2	7.62	4.42	8.65	4.45
4	3.85	2.26	4.35	2.30

For example, the efficiency of the basic solver on 16 nodes is about 0.95, while the efficiency of the reduced solver on 16 nodes is only about 0.70.

A point to stress here is that the reduced MPI solver makes much more efficient use of memory than the basic MPI solver; the basic solver must provide storage for all n positions on each process, while the reduced solver only needs extra storage for $n/$`comm_sz` positions and $n/$`comm_sz` forces. Thus, the extra storage needed on each process for the basic solver is nearly `comm_sz`$/2$ times greater than the storage needed for the reduced solver. When n and `comm_sz` are very large, this factor can easily make the difference between being able to run a simulation only using the process' main memory and having to use secondary storage.

The nodes of the cluster on which we took the timings have four cores, so we can compare the performance of the OpenMP implementations with the performance of the MPI implementations (see Table 6.6). We see that the basic OpenMP solver is a good deal faster than the basic MPI solver. This isn't surprising since `MPI_Allgather` is such an expensive operation. Perhaps surprisingly, though, the reduced MPI solver is quite competitive with the reduced OpenMP solver.

Let's take a brief look at the amount of memory required by the MPI and OpenMP reduced solvers. Say that there are n particles and p threads or processes. Then each solver will allocate the same amount of storage for the local velocities and the local positions. The MPI solver allocates n `doubles` per process for the masses. It also allocates $4n/p$ `doubles` for the `tmp_pos` and `tmp_forces` arrays, so in addition to the

local velocities and positions, the MPI solver stores

$$n + 4n/p$$

`doubles` per process. The OpenMP solver allocates a total of $2pn + 2n$ `doubles` for the forces and n `doubles` for the masses, so in addition to the local velocities and positions, the OpenMP solver stores

$$3n/p + 2n$$

`doubles` per thread. Thus, the difference in the local storage required for the OpenMP version and the MPI version is

$$n - n/p$$

`doubles`. In other words, if n is large, the local storage required for the MPI version is substantially less than the local storage required for the OpenMP version. So, for a fixed number of processes or threads, we should be able to run much larger simulations with the MPI version than the OpenMP version. Of course, because of hardware considerations, we're likely to be able to use many more MPI processes than OpenMP threads, so the size of the largest possible MPI simulations should be *much* greater than the size of the largest possible OpenMP simulations. The MPI version of the reduced solver is much more scalable than any of the other versions, and the "ring pass" algorithm provides a genuine breakthrough in the design of n-body solvers.

6.2 TREE SEARCH

Many problems can be solved using a tree search. As a simple example, consider the traveling salesperson problem, or TSP. In TSP, a salesperson is given a list of cities she needs to visit and a cost for traveling between each pair of cities. Her problem is to visit each city once, returning to her hometown, and she must do this with the least possible cost. A route that starts in her hometown, visits each city once and returns to her hometown is called a *tour*; thus, the TSP is to find a minimum-cost tour.

Unfortunately, TSP is what's known as an **NP-complete** problem. From a practical standpoint, this means that there is no algorithm known for solving it that, in all cases, is significantly better than exhaustive search. Exhaustive search means examining all possible solutions to the problem and choosing the best. The number of possible solutions to TSP grows exponentially as the number of cities is increased. For example, if we add one additional city to an n-city problem, we'll increase the number of possible solutions by a factor of $n - 1$. Thus, although there are only six possible solutions to a four-city problem, there are $4 \times 6 = 24$ to a five-city problem, $5 \times 24 = 120$ to a six-city problem, $6 \times 120 = 720$ to a seven-city problem, and so on. In fact, a 100-city problem has far more possible solutions than the number of atoms in the universe!

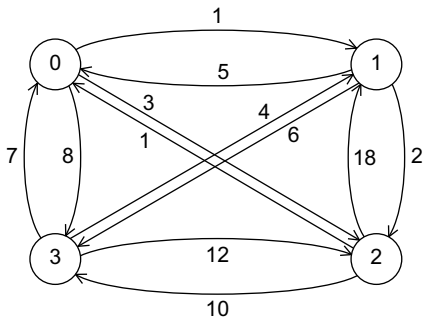


FIGURE 6.9

A four-city TSP

Furthermore, if we could find a solution to TSP that's significantly better in all cases than exhaustive search, then there are literally hundreds of other very hard problems for which we could find fast solutions. Not only is there no known solution to TSP that is better in all cases than exhaustive search, it's very unlikely that we'll find one.

So how can we solve TSP? There are a number of clever solutions. However, let's take a look at an especially simple one. It's a very simple form of tree search. The idea is that in searching for solutions, we build a *tree.* The leaves of the tree correspond to tours, and the other tree nodes correspond to "partial" tours—routes that have visited some, but not all, of the cities.

Each node of the tree has an associated cost, that is, the cost of the partial tour. We can use this to eliminate some nodes of the tree. Thus, we want to keep track of the cost of the best tour so far, and, if we find a partial tour or node of the tree that couldn't possibly lead to a less expensive complete tour, we shouldn't bother searching the children of that node (see Figures 6.9 and 6.10).

In Figure 6.9 we've represented a four-city TSP as a labeled, directed graph. A **graph** (not to be confused with a graph in calculus) is a collection of vertices and edges or line segments joining pairs of vertices. In a **directed graph** or **digraph**, the edges are oriented—one end of each edge is the tail, and the other is the head. A graph or digraph is **labeled** if the vertices and/or edges have labels. In our example, the vertices of the digraph correspond to the cities in an instance of the TSP, the edges correspond to routes between the cities, and the labels on the edges correspond to the costs of the routes. For example, there's a cost of 1 to go from city 0 to city 1 and a cost of 5 to go from city 1 to city 0.

If we choose vertex 0 as the salesperson's home city, then the initial partial tour consists only of vertex 0, and since we've gone nowhere, it's cost is 0. Thus, the root of the tree in Figure 6.10 has the partial tour consisting only of the vertex 0 with cost 0. From 0 we can first visit 1, 2, or 3, giving us three two-city partial tours with costs 1, 3, and 8, respectively. In Figure 6.10 this gives us three children of the root. Continuing, we get six three-city partial tours, and since there are

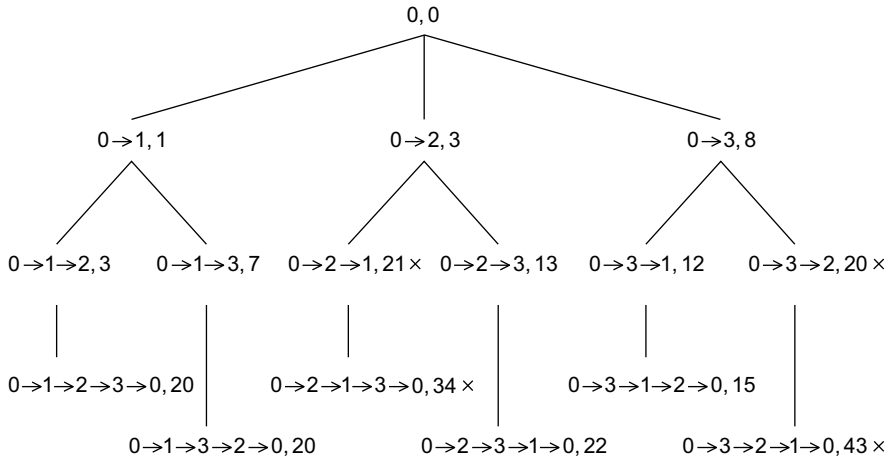


FIGURE 6.10

Search tree for four-city TSP

only four cities, once we've chosen three of the cities, we know what the complete tour is.

Now, to find a least-cost tour, we should search the tree. There are many ways to do this, but one of the most commonly used is called **depth-first search**. In depth-first search, we probe as deeply as we can into the tree. After we've either reached a leaf or found a tree node that can't possibly lead to a least-cost tour, we back up to the deepest "ancestor" tree node with unvisited children, and probe one of its children as deeply as possible.

In our example, we'll start at the root, and branch left until we reach the leaf labeled

$$0 \to 1 \to 2 \to 3 \to 0, \text{ Cost } 20.$$

Then we back up to the tree node labeled $0 \to 1$, since it is the deepest ancestor node with unvisited children, and we'll branch down to get to the leaf labeled

$$0 \to 1 \to 3 \to 2 \to 0, \text{ Cost } 20.$$

Continuing, we'll back up to the root and branch down to the node labeled $0 \to 2$. When we visit its child, labeled

$$0 \to 2 \to 1, \text{ Cost } 21,$$

we'll go no further in this subtree, since we've already found a complete tour with cost less than 21. We'll back up to $0 \to 2$ and branch down to its remaining unvisited child. Continuing in this fashion, we eventually find the least-cost tour

$$0 \to 3 \to 1 \to 2 \to 0, \text{ Cost } 15.$$

6.2.1 Recursive depth-first search

Using depth-first search we can systematically visit each node of the tree that could possibly lead to a least-cost solution. The simplest formulation of depth-first search uses recursion (see Program 6.4). Later on it will be useful to have a definite order in which the cities are visited in the `for` loop in Lines 8 to 13, so we'll assume that the cities are visited in order of increasing index, from city 1 to city $n - 1$.

The algorithm makes use of several global variables:

- `n`: the total number of cities in the problem
- `digraph`: a data structure representing the input digraph
- `hometown`: a data structure representing vertex or city 0, the salesperson's hometown
- `best_tour`: a data structure representing the best tour so far

The function `City_count` examines the partial tour `tour` to see if there are n cities on the partial tour. If there are, we know that we simply need to return to the hometown to complete the tour, and we can check to see if the complete tour has a lower cost than the current "best tour" by calling `Best_tour`. If it does, we can replace the current best tour with this tour by calling the function `Update_best_tour`. Note that before the first call to `Depth_first_search`, the `best_tour` variable should be initialized so that its cost is greater than the cost of any possible least-cost tour.

If the partial tour `tour` hasn't visited n cities, we can continue branching down in the tree by "expanding the current node," in other words, by trying to visit other cities from the city last visited in the partial tour. To do this we simply loop through the cities. The function `Feasible` checks to see if the city or vertex has already been visited, and, if not, whether it can possibly lead to a least-cost tour. If the city is feasible, we add it to the tour, and recursively call `Depth_first_search`. When

```
1  void Depth_first_search(tour_t tour) {
2     city_t city;
3
4     if (City_count(tour) == n) {
5        if (Best_tour(tour))
6           Update_best_tour(tour);
7     } else {
8        for each neighboring city
9           if (Feasible(tour, city)) {
10             Add_city(tour, city);
11             Depth_first_search(tour);
12             Remove_last_city(tour, city);
13          }
14    }
15 }  /* Depth_first_search */
```

Program 6.4: Pseudocode for a recursive solution to TSP using depth-first search

we return from Depth_first_search, we remove the city from the tour, since it shouldn't be included in the tour used in subsequent recursive calls.

6.2.2 Nonrecursive depth-first search

Since function calls are expensive, recursion can be slow. It also has the disadvantage that at any given instant of time only the current tree node is accessible. This could be a problem when we try to parallelize tree search by dividing tree nodes among the threads or processes.

It is possible to write a nonrecursive depth-first search. The basic idea is modeled on recursive implementation. Recall that recursive function calls can be implemented by pushing the current state of the recursive function onto the run-time stack. Thus, we can try to eliminate recursion by pushing necessary data on our own stack before branching deeper into the tree, and when we need to go back up the tree—either because we've reached a leaf or because we've found a node that can't lead to a better solution—we can pop the stack.

This outline leads to the implementation of iterative depth-first search shown in Program 6.5. In this version, a stack record consists of a single city, the city that will be added to the tour when its record is popped. In the recursive version we continue to make recursive calls until we've visited every node of the tree that corresponds to a feasible partial tour. At this point, the stack won't have any more activation records for calls to Depth_first_search, and we'll return to the function that made the

```
1  for (city = n-1; city >= 1; city--)
2     Push(stack, city);
3  while (!Empty(stack)) {
4     city = Pop(stack);
5     if (city == NO_CITY) // End of child list, back up
6        Remove_last_city(curr_tour);
7     else {
8        Add_city(curr_tour, city);
9        if (City_count(curr_tour) == n) {
10          if (Best_tour(curr_tour))
11             Update_best_tour(curr_tour);
12          Remove_last_city(curr_tour);
13       } else {
14          Push(stack, NO_CITY);
15          for (nbr = n-1; nbr >= 1; nbr--)
16             if (Feasible(curr_tour, nbr))
17                Push(stack, nbr);
18       }
19    }  /* if Feasible */
20 }  /* while !Empty */
```

Program 6.5: Pseudocode for an implementation of a depth-first solution to TSP that doesn't use recursion

original call to `Depth_first_search`. The main control structure in our iterative version is the `while` loop extending from Line 3 to Line 20, and the loop termination condition is that our stack is empty. As long as the search needs to continue, we need to make sure the stack is nonempty, and, in the first two lines, we add each of the non-hometown cities. Note that this loop visits the cities in decreasing order, from $n - 1$ down to 1. This is because of the order created by the stack, whereby the stack pops the top cities first. By reversing the order, we can insure that the cities are visited in the same order as the recursive function.

Also notice that in Line 5 we check whether the city we've popped is the constant `NO_CITY`. This constant is used so that we can tell when we've visited all of the children of a tree node; if we didn't use it, we wouldn't be able to tell when to back up in the tree. Thus, before pushing all of the children of a node (Lines 15–17), we push the `NO_CITY` marker.

An alternative to this iterative version uses partial tours as stack records (see Program 6.6). This gives code that is closer to the recursive function. However, it also results in a slower version, since it's necessary for the function that pushes onto the stack to create a copy of the tour before actually pushing it on to the stack. To emphasize this point, we've called the function `Push_copy`. (What happens if we simply push a pointer to the current tour onto the stack?) The extra memory required will probably not be a problem. However, allocating storage for a new tour and copying the existing tour is time-consuming. To some degree we can mitigate these costs by saving freed tours in our own data structure, and when a freed tour is available we can use it in the `Push_copy` function instead of calling `malloc`.

On the other hand, this version has the virtue that the stack is more or less independent of the other data structures. Since entire tours are stored, multiple threads or processes can "help themselves" to tours, and, if this is done reasonably carefully,

```
 1  Push_copy(stack, tour);  // Tour that visits only the hometown
 2  while (!Empty(stack)) {
 3     curr_tour = Pop(stack);
 4     if (City_count(curr_tour) == n) {
 5        if (Best_tour(curr_tour))
 6           Update_best_tour(curr_tour);
 7     } else {
 8        for (nbr = n-1; nbr >= 1; nbr--)
 9           if (Feasible(curr_tour, nbr)) {
10              Add_city(curr_tour, nbr);
11              Push_copy(stack, curr_tour);
12              Remove_last_city(curr_tour);
13           }
14     }
15     Free_tour(curr_tour);
16  }
```

Program 6.6: Pseudocode for a second solution to TSP that doesn't use recursion

it won't destroy the correctness of the program. With the original iterative version, a stack record is just a city and it doesn't provide enough information by itself to show where we are in the tree.

6.2.3 Data structures for the serial implementations

Our principal data structures are the tour, the digraph, and, in the iterative implementations, the stack. The tour and the stack are essentially list structures. In problems that we're likely to be able to tackle, the number of cities is going to be small—certainly less than 100—so there's no great advantage to using a linked list to represent the tours and we've used an array that can store $n+1$ cities. We repeatedly need both the number of cities in the partial tour and the cost of the partial tour. Therefore, rather than just using an array for the tour data structure and recomputing these values, we use a struct with three members: the array storing the cities, the number of cities, and the cost of the partial tour.

To improve the readability and the performance of the code, we can use preprocessor macros to access the members of the struct. However, since macros can be a nightmare to debug, it's a good idea to write "accessor" functions for use during initial development. When the program with accessor functions is working, they can be replaced with macros. As an example, we might start with the function

```
/* Find the ith city on the partial tour */
int Tour_city(tour_t tour, int i) {
   return tour->cities[i];
}  /* Tour_city */
```

When the program is working, we could replace this with the macro

```
/* Find the ith city on the partial tour */
#define Tour_city(tour, i) (tour->cities[i])
```

The stack in the original iterative version is just a list of cities or `int`s. Furthermore, since there can't be more than $n^2/2$ records on the stack (see Exercise 6.17) at any one time, and n is likely to be small, we can just use an array, and like the tour data structure, we can store the number of elements on the stack. Thus, for example, `Push` can be implemented with

```
void Push(my_stack_t stack, int city) {
   int loc = stack->list_sz;
   stack->list[loc] = city;
   stack->list_sz++;
}  /* Push */
```

In the second iterative version, the version that stores entire tours in the stack, we can probably still use an array to store the tours on the stack. Now the push function will look something like this:

```
void Push_copy(my_stack_t stack, tour_t tour) {
   int loc = stack->list_sz;
```

```
    tour_t tmp = Alloc_tour();
    Copy_tour(tour, tmp);
    stack->list[loc] = tmp;
    stack->list_sz++;
}  /* Push */
```

Once again, element access for the stack can be implemented with macros.

There are many possible representations for digraphs. When the digraph has relatively few edges, list representations are preferred. However, in our setting, if vertex i is different from vertex j, there are directed, weighted edges from i to j and from j to i, so we need to store a weight for each ordered pair of distinct vertices i and j. Thus, in our setting, an **adjacency matrix** is almost certainly preferable to a list structure. This is an $n \times n$ matrix, in which the weight of the edge from vertex i to vertex j can be the entry in the ith row and jth column of the matrix. We can access this weight directly, without having to traverse a list. The diagonal elements (row i and column i) aren't used, and we'll set them to 0.

6.2.4 Performance of the serial implementations

The run-times of the three serial implementations are shown in Table 6.7. The input digraph contained 15 vertices (including the hometown), and all three algorithms visited approximately 95,000,000 tree nodes. The first iterative version is less than 5% faster than the recursive version, and the second iterative version is about 8% slower than the recursive version. As expected, the first iterative solution eliminates some of the overhead due to repeated function calls, while the second iterative solution is slower because of the repeated copying of tour data structures. However, as we'll see, the second iterative solution is relatively easy to parallelize, so we'll be using it as the basis for the parallel versions of tree search.

6.2.5 Parallelizing tree search

Let's take a look at parallelizing tree search. The tree structure suggests that we identify tasks with tree nodes. If we do this, the tasks will communicate down the tree edges: a parent will communicate a new partial tour to a child, but a child, except for terminating, doesn't communicate directly with a parent.

We also need to take into consideration the updating and use of the best tour. Each task examines the best tour to determine whether the current partial tour is feasible or the current complete tour has lower cost. If a leaf task determines its tour is a better tour, then it will also update the best tour. Although all of the actual computation can

Table 6.7 Run-Times of the Three Serial Implementations of Tree Search (times in seconds)

Recursive	First Iterative	Second Iterative
30.5	29.2	32.9

be considered to be carried out by the tree node tasks, we need to keep in mind that the best tour data structure requires additional communication that is not explicit in the tree edges. Thus, it's convenient to add an additional task that corresponds to the best tour. It "sends" data to every tree node task, and receives data from some of the leaves. This latter view is convenient for shared-memory, but not so convenient for distributed-memory.

A natural way to agglomerate and map the tasks is to assign a subtree to each thread or process, and have each thread/process carry out all the tasks in its subtree. For example, if we have three threads or processes, as shown earlier in Figure 6.10, we might map the subtree rooted at $0 \rightarrow 1$ to thread/process 0, the subtree rooted at $0 \rightarrow 2$ to thread/process 1, and the subtree rooted at $0 \rightarrow 3$ to thread/process 2.

Mapping details

There are many possible algorithms for identifying which subtrees we assign to the processes or threads. For example, one thread or process could run the last version of serial depth-first search until the stack stores one partial tour for each thread or process. Then it could assign one tour to each thread or process. The problem with depth-first search is that we expect a subtree whose root is deeper in the tree to require less work than a subtree whose root is higher up in the tree, so we would probably get better load balance if we used something like **breadth-first search** to identify the subtrees.

As the name suggests, breadth-first search searches as widely as possible in the tree before going deeper. So if, for example, we carry out a breadth-first search until we reach a level of the tree that has at least `thread_count` or `comm_sz` nodes, we can then divide the nodes at this level among the threads or processes. See Exercise 6.18 for implementation details.

The best tour data structure

On a shared-memory system, the best tour data structure can be shared. In this setting, the `Feasible` function can simply examine the data structure. However, updates to the best tour will cause a race condition, and we'll need some sort of locking to prevent errors. We'll discuss this in more detail when we implement the parallel version.

In the case of a distributed-memory system, there are a couple of choices that we need to make about the best tour. The simplest option would be to have the processes operate independently of each other until they have completed searching their subtrees. In this setting, each process would store its own *local* best tour. This local best tour would be used by the process in `Feasible` and updated by the process each time it calls `Update_best_tour`. When all the processes have finished searching, they can perform a global reduction to find the tour with the *global* least cost.

This approach has the virtue of simplicity, but it also suffers from the problem that it's entirely possible for a process to spend most or all of its time searching through partial tours that couldn't possibly lead to a global best tour. Thus, we should

probably try using an approach that makes the current global best tour available to all the processes. We'll take a look at details when we discuss the MPI implementation.

Dynamic mapping of tasks

A second issue we should consider is the problem of load imbalance. Although the use of breadth-first search ensures that all of our subtrees have approximately the same number of nodes, there is no guarantee that they all have the same amount of work. It's entirely possible that one process or thread will have a subtree consisting of very expensive tours, and, as a consequence, it won't need to search very deeply into its assigned subtree. However, with our current, *static* mapping of tasks to threads/processes, this one thread or process will simply have to wait until the other threads/processes are done.

An alternative is to implement a **dynamic** mapping scheme. In a dynamic scheme, if one thread/process runs out of useful work, it can obtain additional work from another thread/process. In our final implementation of serial depth-first search, each stack record contains a partial tour. With this data structure a thread or process can give additional work to another thread/process by dividing the contents of its stack. This might at first seem to have the potential for causing problems with the program's correctness, since if we give part of one thread's or one process' stack to another, there's a good chance that the order in which the tree nodes will be visited will be changed.

However, we're already going to do this; when we assign different subtrees to different threads/processes, the order in which the tree nodes are visited is no longer the serial depth-first ordering. In fact, in principle, there's no reason to visit any node before any other node as long as we make sure we visit "ancestors" before "descendants." But this isn't a problem since a partial tour isn't added to the stack until after all its ancestors have been visited. For example, in Figure 6.10 the node consisting of the tour $0 \rightarrow 2 \rightarrow 1$ will be pushed onto the stack when the node consisting of the tour $0 \rightarrow 2$ is the currently active node, and consequently the two nodes won't be on the stack simultaneously. Similarly, the parent of $0 \rightarrow 2$, the root of the tree, 0, is no longer on the stack when $0 \rightarrow 2$ is visited.

A second alternative for dynamic load balancing—at least in the case of shared memory—would be to have a shared stack. However, we couldn't simply dispense with the local stacks. If a thread needed to access the shared stack every time it pushed or popped, there would be a tremendous amount of contention for the shared stack and the performance of the program would probably be worse than a serial program. This is exactly what happened when we parallelized the reduced n-body solver with mutexes/locks protecting the calculations of the total forces on the various particles. If every call to `Push` or `Pop` formed a critical section, our program would grind to nearly a complete halt. Thus, we would want to retain local stacks for each thread, with only occasional accesses to the shared stack. We won't pursue this alternative. See Programming Assignment 6.7 for further details.

6.2.6 A static parallelization of tree search using pthreads

In our static parallelization, a single thread uses breadth-first search to generate enough partial tours so that each thread gets at least one partial tour. Then each thread takes its partial tours and runs iterative tree search on them. We can use the pseudocode shown in Program 6.7 on each thread. Note that most of the function calls—for example, `Best_tour`, `Feasible`, `Add_city`—need to access the adjacency matrix representing the digraph, so all the threads will need to access the digraph. However, since these are only *read* accesses, this won't result in a race condition or contention among the threads.

There are only four potential differences between this pseudocode and the pseudocode we used for the second iterative serial implementation:

- The use of `my_stack` instead of `stack`; since each thread has its own, private stack, we use `my_stack` as the identifier for the stack object instead of `stack`.
- Initialization of the stack.
- Implementation of the `Best_tour` function.
- Implementation of the `Update_best_tour` function.

In the serial implementation, the stack is initialized by pushing the partial tour consisting only of the hometown onto the stack. In the parallel version we need to generate at least `thread_count` partial tours to distribute among the threads. As we discussed earlier, we can use breadth-first search to generate a list of at least `thread_count` tours by having a single thread search the tree until it reaches a level with at least `thread_count` tours. (Note that this implies that the number of threads should be less than $(n-1)!$, which shouldn't be a problem). Then the threads can

```
Partition_tree(my_rank, my_stack);

while (!Empty(my_stack)) {
   curr_tour = Pop(my_stack);
   if (City_count(curr_tour) == n) {
      if (Best_tour(curr_tour)) Update_best_tour(curr_tour);
   } else {
      for (city = n-1; city >= 1; city--)
         if (Feasible(curr_tour, city)) {
            Add_city(curr_tour, city);
            Push_copy(my_stack, curr_tour);
            Remove_last_city(curr_tour)
         }
   }
   Free_tour(curr_tour);
}
```

Program 6.7: Pseudocode for a Pthreads implementation of a statically parallelized solution to TSP

use a block partition to divide these tours among themselves and push them onto their private stacks. Exercise 6.18 looks into the details.

To implement the `Best_tour` function, a thread should compare the cost of its current tour with the cost of the global best tour. Since multiple threads may be simultaneously accessing the global best cost, it might at first seem that there will be a race condition. However, the `Best_tour` function only *reads* the global best cost, so there won't be any conflict with threads that are also checking the best cost. If a thread is updating the global best cost, then a thread that is just checking it will either read the old value or the new, updated value. While we would prefer that it get the new value, we can't insure this without using some very costly locking strategy. For example, threads wanting to execute `Best_tour` or `Update_best_tour` could wait on a single mutex. This would insure that no thread is updating while another thread is only checking, but would have the unfortunate side effect that only one thread could check the best cost at a time. We could improve on this by using a read-write lock, but this would have the side effect that the readers—the threads calling `Best_tour`—would all block while a thread updated the best tour. In principle, this doesn't sound too bad, but recall that in practice read-write locks can be quite slow. So it seems pretty clear that the "no contention" solution of possibly getting a best tour cost that's out-of-date is probably better, as the next time the thread calls `Best_tour`, it will get the updated value of the best tour cost.

On the other hand, we call `Update_best_tour` with the intention of *writing* to the best tour structure, and this clearly can cause a race condition if two threads call it simultaneously. To avoid this problem, we can protect the body of the `Update_best_tour` function with a mutex. This isn't enough, however; between the time a thread completes the test in `Best_tour` and the time it obtains the lock in `Update_best_tour`, another thread may have obtained the lock and updated the best tour cost, which now may be less than the best tour cost that the first thread found in `Best_tour`. Thus, correct pseudocode for `Update_best_tour` should look something like this:

```
pthread_mutex_lock(best_tour_mutex);
/* We've already checked Best_tour, but we need to check it
    again */
if (Best_tour(tour))
   Replace old best tour with tour;
pthread_mutex_unlock(best_tour_mutex).
```

This may seem wasteful, but if updates to the best tour are infrequent, then most of the time `Best_tour` will return `false` and it will only be rarely necessary to make the "double" call.

6.2.7 A dynamic parallelization of tree search using pthreads

If the initial distribution of subtrees doesn't do a good job of distributing the work among the threads, the static parallelization provides no means of redistributing work. The threads with "small" subtrees will finish early, while the threads with large subtrees will continue to work. It's not difficult to imagine that one thread gets the lion's

share of the work because the edges in its initial tours are very cheap, while the edges in the other threads' initial tours are very expensive. To address this issue, we can try to dynamically redistribute the work as the computation proceeds.

To do this, we can replace the test `!Empty(my_stack)` controlling execution of the `while` loop with more complex code. The basic idea is that when a thread runs out of work—that is, `!Empty(my_stack)` becomes false—instead of immediately exiting the `while` loop, the thread waits to see if another thread can provide more work. On the other hand, if a thread that still has work in its stack finds that there is at least one thread without work, and its stack has at least two tours, it can "split" its stack and provide work for one of the threads.

Pthreads condition variables provide a natural way to implement this. When a thread runs out of work it can call `pthread_cond_wait` and go to sleep. When a thread with work finds that there is at least one thread waiting for work, after splitting its stack, it can call `pthread_cond_signal`. When a thread is awakened it can take one of the halves of the split stack and return to work.

This idea can be extended to handle termination. If we maintain a count of the number of threads that are in `pthread_cond_wait`, then when a thread whose stack is empty finds that `thread_count` − 1 threads are already waiting, it can call `pthread_cond_broadcast` and as the threads awaken, they'll see that all the threads have run out of work and quit.

Termination

Thus, we can use the pseudocode shown in Program 6.8 for a `Terminated` function that would be used instead of `Empty` for the `while` loop implementing tree search.

There are several details that we should look at more closely. Notice that the code executed by a thread before it splits its stack is fairly complicated. In Lines 1–2 the thread

- checks that it has at least two tours in its stack,
- checks that there are threads waiting, and
- checks whether the `new_stack` variable is `NULL`.

The reason for the check that the thread has enough work should be clear: if there are fewer than two records on the thread's stack, "splitting" the stack will either do nothing or result in the active thread's trading places with one of the waiting threads.

It should also be clear that there's no point in splitting the stack if there aren't any threads waiting for work. Finally, if some thread has already split its stack, but a waiting thread hasn't retrieved the new stack, that is, `new_stack != NULL`, then it would be disastrous to split a stack and overwrite the existing new stack. Note that this makes it essential that after a thread retrieves `new_stack` by, say, copying `new_stack` into its private `my_stack` variable, the thread must set `new_stack` to `NULL`.

If all three of these conditions hold, then we can try splitting our stack. We can acquire the mutex that protects access to the objects controlling termination (`threads_in_cond_wait`, `new_stack`, and the condition variable). However, the condition

```
threads_in_cond_wait > 0 && new_stack == NULL
```

```
1  if (my_stack_size >= 2 && threads_in_cond_wait > 0 &&
2        new_stack == NULL) {
3     lock term_mutex;
4     if (threads_in_cond_wait > 0 && new_stack == NULL) {
5        Split my_stack creating new_stack;
6        pthread_cond_signal(&term_cond_var);
7     }
8     unlock term_mutex;
9     return 0;  /* Terminated = false; don't quit */
10 } else if (!Empty(my_stack)) /* Keep working */
11    return 0;  /* Terminated = false; don't quit */
12 } else { /* My stack is empty */
13    lock term_mutex;
14    if (threads_in_cond_wait == thread_count-1)
15                                        /* Last thread running */
16       threads_in_cond_wait++;
17       pthread_cond_broadcast(&term_cond_var);
18       unlock term_mutex;
19       return 1;  /* Terminated = true; quit */
20    } else { /* Other threads still working, wait for work */
21       threads_in_cond_wait++;
22       while (pthread_cond_wait(&term_cond_var, &term_mutex) != 0);
23       /* We've been awakened */
24       if (threads_in_cond_wait < thread_count) { /* We got work */
25          my_stack = new_stack;
26          new_stack = NULL;
27          threads_in_cond_wait--;
28          unlock term_mutex;
29          return 0;  /* Terminated = false */
30       } else {  /* All threads done */
31          unlock term_mutex;
32          return 1;  /* Terminated = true; quit */
33       }
34    }  /* else wait for work */
35 }  /* else my_stack is empty */
```

Program 6.8: Pseudocode for Pthreads `Terminated` function

can change between the time we start waiting for the mutex and the time we actually acquire it, so as with `Update_best_tour`, we need to confirm that this condition is still true after acquiring the mutex (Line 4). Once we've verified that these conditions still hold, we can split the stack, awaken one of the waiting threads, unlock the mutex, and return to work.

If the test in Lines 1 and 2 is false, we can check to see if we have any work at all—that is, our stack is nonempty. If it is, we return to work. If it isn't, we'll start the termination sequence by waiting for and acquiring the termination mutex in Line 13. Once we've acquired the mutex, there are two possibilities:

- We're the last thread to enter the termination sequence, that is, `threads_in_cond_wait == thread_count-1`.
- Other threads are still working.

In the first case, we know that since all the other threads have run out of work, and we have also run out of work, the tree search should terminate. We therefore signal all the other threads by calling `pthread_cond_broadcast` and returning true. Before executing the broadcast, we increment `threads_in_cond_wait`, even though the broadcast is telling all the threads to return from the condition wait. The reason is that `threads_in_cond_wait` is serving a dual purpose: When it's less than `thread_count`, it tells us how many threads are waiting. However, when it's equal to `thread_count`, it tells us that all the threads are out of work, and it's time to quit.

In the second case—other threads are still working—we call `pthread_cond_wait` (Line 22) and wait to be awakened. Recall that it's possible that a thread could be awakened by some event other than a call to `pthread_cond_signal` or `pthread_cond_broadcast`. So, as usual, we put the call to `pthread_cond_wait` in a `while` loop, which will immediately call `pthread_cond_wait` again if some other event (return value not 0) awakens the thread.

Once we've been awakened, there are also two cases to consider:

- `threads_in_cond_wait < thread_count`
- `threads_in_cond_wait == thread_count`

In the first case, we know that some other thread has split its stack and created more work. We therefore copy the newly created stack into our private stack, set the `new_stack` variable to `NULL`, and decrement `threads_in_cond_wait` (i.e., Lines 25–27). Recall that when a thread returns from a condition wait, it obtains the mutex associated with the condition variable, so before returning, we also unlock the mutex (i.e., Line 28). In the second case, there's no work left, so we unlock the mutex and return true.

In the actual code, we found it convenient to group the termination variables together into a single struct. Thus, we defined something like

```
typedef struct {
   my_stack_t new_stack;
   int threads_in_cond_wait;
   pthread_cond_t term_cond_var;
   pthread_mutex_t term_mutex;
} term_struct;
typedef term_struct* term_t;

term_t term;  // global variable
```

and we defined a couple of functions, one for initializing the `term` variable and one for destroying/freeing the variable and its members.

Before discussing the function that splits the stack, note that it's possible that a thread with work can spend a lot of time waiting for `term_mutex` before being able

to split its stack. Other threads may be either trying to split their stacks, or preparing for the condition wait. If we suspect that this is a problem, Pthreads provides a nonblocking alternative to `pthread_mutex_lock` called `pthread_mutex_trylock`:

```
int pthread_mutex_trylock(
          pthread_mutex_t*   mutex_p   /* in/out */);
```

This function attempts to acquire `mutex_p`. However, if it's locked, instead of waiting, it returns immediately. The return value will be zero if the calling thread has successfully acquired the mutex, and nonzero if it hasn't. As an alternative to waiting on the mutex before splitting its stack, a thread can call `pthread_mutex_trylock`. If it acquires `term_mutex`, it can proceed as before. If not, it can just return. Presumably on a subsequent call it can successfully acquire the mutex.

Splitting the stack

Since our goal is to balance the load among the threads, we would like to insure that the amount of work in the new stack is roughly the same as the amount remaining in the original stack. We have no way of knowing in advance of searching the subtree rooted at a partial tour how much work is actually associated with the partial tour, so we'll never be able to guarantee an equal division of work. However, we can use the same strategy that we used in our original assignment of subtrees to threads: that the subtrees rooted at two partial tours with the same number of cities have identical structures. Since on average two partial tours with the same number of cities are equally likely to lead to a "good" tour (and hence more work), we can try splitting the stack by assigning the tours on the stack on the basis of their numbers of edges. The tour with the least number of edges remains on the original stack, the tour with the next to the least number of edges goes to the new stack, the tour with the next number of edges remains on the original, and so on.

This is fairly simple to implement, since the tours on the stack have an increasing number of edges. That is, as we proceed from the bottom of the stack to the top of the stack, the number of edges in the tours increases. This is because when we push a new partial tour with k edges onto the stack, the tour that's immediately "beneath" it on the stack either has k edges or $k - 1$ edges. We can implement the split by starting at the bottom of the stack, and alternately leaving partial tours on the old stack and pushing partial tours onto the new stack, so tour 0 will stay on the old stack, tour 1 will go to the new stack, tour 2 will stay on the old stack, and so on. If the stack is implemented as an array of tours, this scheme will require that the old stack be "compressed" so that the gaps left by removing alternate tours are eliminated. If the stack is implemented as a linked list of tours, compression won't be necessary.

This scheme can be further refined by observing that partial tours with lots of cities won't provide much work, since the subtrees that are rooted at these trees are very small. We could add a "cutoff size" and not reassign a tour unless its number of cities was less than the cutoff. In a shared-memory setting with an array-based stack, reassigning a tour when a stack is split won't increase the cost of the split, since the tour (which is a pointer) will either have to be copied to the new stack or a new

Table 6.8 Run-Times of Pthreads Tree-Search Programs (times in seconds)

	First Problem				Second Problem			
Threads	*Serial*	*Static*	*Dynamic*		*Serial*	*Static*	*Dynamic*	
1	32.9	32.7	34.7	(0)	26.0	25.8	27.5	(0)
2		27.9	28.9	(7)		25.8	19.2	(6)
4		25.7	25.9	(47)		25.8	9.3	(49)
8		23.8	22.4	(180)		24.0	5.7	(256)

location in the old stack. We'll defer exploration of this alternative to Programming Assignment 6.6.

6.2.8 Evaluating the Pthreads tree-search programs

Table 6.8 shows the performance of the two Pthreads programs on two fifteen-city problems. The "Serial" column gives the run-time of the second iterative solution—the solution that pushes a copy of each new tour onto the stack. For reference, the first problem in Table 6.8 is the same as the problem the three serial solutions were tested with in Table 6.7, and both the Pthreads and serial implementations were tested on the same system. Run-times are in seconds, and the numbers in parentheses next to the run-times of the program that uses dynamic partitioning give the total number of times the stacks were split.

From these numbers, it's apparent that different problems can result in radically different behaviors. For example, the program that uses static partitioning generally performs better on the first problem than the program that uses dynamic partitioning. However, on the second problem, the performance of the static program is essentially independent of the number of threads, while the dynamic program obtains excellent performance. In general, it appears that the dynamic program is more scalable than the static program.

As we increase the number of threads, we would expect that the size of the local stacks will decrease, and hence threads will run out of work more often. When threads are waiting, other threads will split their stacks, so as the number of threads is increased, the total number of stack splits should increase. Both problems confirm this prediction.

It should be noted that if the input problem has more than one possible solution—that is, different tours with the same minimum cost—then the results of both of the programs are nondeterministic. In the static program, the sequence of best tours depends on the speed of the threads, and this sequence determines which tree nodes are examined. In the dynamic program, we also have nondeterminism because different runs may result in different places where a thread splits its stack and variation in which thread receives the new work. This can also result in run-times, especially dynamic run-times, that are *highly* variable.

6.2.9 Parallelizing the tree-search programs using OpenMP

The issues involved in implementing the static and dynamic parallel tree-search programs using OpenMP are the same as the issues involved in implementing the programs using Pthreads.

There are almost no substantive differences between a static implementation that uses OpenMP and one that uses Pthreads. However, a couple of points should be mentioned:

1. When a single thread executes some code in the Pthreads version, the test

   ```
   if (my_rank == whatever)
   ```

   can be replaced by the OpenMP directive

   ```
   #  pragma omp single
   ```

   This will insure that the following structured block of code will be executed by one thread in the team, and the other threads in the team will wait in an implicit barrier at the end of the block until the executing thread is finished.

   When `whatever` is 0 (as it is in each test in the Pthreads program), the test can also be replaced by the OpenMP directive

   ```
   #  pragma omp master
   ```

   This will insure that thread 0 executes the following structured block of code. However, the `master` directive doesn't put an implicit barrier at the end of the block, so it may be necessary to also add a `barrier` directive after a structured block that has been modified by a `master` directive.
2. The Pthreads mutex that protects the best tour can be replaced by a single `critical` directive placed either inside the `Update_best_tour` function or immediately before the call to `Update_best_tour`. This is the only potential source of a race condition after the distribution of the initial tours, so the simple `critical` directive won't cause a thread to block unnecessarily.

The dynamically load-balanced Pthreads implementation depends heavily on Pthreads condition variables, and OpenMP doesn't provide a comparable object. The rest of the Pthreads code can be easily converted to OpenMP. In fact, OpenMP even provides a nonblocking version of `omp_set_lock`. Recall that OpenMP provides a lock object `omp_lock_t` and the following functions for acquiring and relinquishing the lock, respectively:

```
void omp_set_lock(omp_lock_t*    lock_p   /* in/out */);
void omp_unset_lock(omp_lock_t*  lock_p   /* in/out */);
```

It also provides the function

```
int omp_test_lock(omp_lock_t*    lock_p   /* in/out */);
```

which is analogous to pthread_mutex_trylock; it attempts to acquire the lock *lock_p, and if it succeeds it returns true (or nonzero). If the lock is being used by some other thread, it returns immediately with return value false (or zero).

If we examine the pseudocode for the Pthreads Terminated function in Program 6.8, we see that in order to adapt the Pthreads version to OpenMP, we need to emulate the functionality of the Pthreads function calls

```
pthread_cond_signal(&term_cond_var);
pthread_cond_broadcast(&term_cond_var);
pthread_cond_wait(&term_cond_var, &term_mutex);
```

in Lines 6, 17, and 22, respectively.

Recall that a thread that has entered the condition wait by calling

```
pthread_cond_wait(&term_cond_var, &term_mutex);
```

is waiting for either of two events:

- Another thread has split its stack and created work for the waiting thread.
- All of the threads have run out of work.

Perhaps the simplest solution to emulating a condition wait in OpenMP is to use busy-waiting. Since there are two conditions a waiting thread should test for, we can use two different variables in the busy-wait loop:

```
/* Global variables */
int awakened_thread = -1;
int work_remains = 1;  /* true */
. . .
while (awakened_thread != my_rank && work_remains);
```

Initialization of the two variables is crucial: If awakened_thread has the value of some thread's rank, that thread will exit immediately from the while, but there may be no work available. Similarly, if work_remains is initialized to 0, all the threads will exit the while loop immediately and quit.

Now recall that when a thread enters a Pthreads condition wait, it relinquishes the mutex associated with the condition variable so that another thread can also enter the condition wait or signal the waiting thread. Thus, we should relinquish the lock used in the Terminated function before starting the while loop.

Also recall that when a thread returns from a Pthreads condition wait, it reacquires the mutex associated with the condition variable. This is especially important in this setting since if the awakened thread has received work, it will need to access the shared data structures storing the new stack. Thus, our complete emulated condition wait should look something like this:

```
/* Global vars */
int awakened_thread = -1;
work_remains = 1;  /* true */
. . .
```

```
omp_unset_lock(&term_lock);
while (awakened_thread != my_rank && work_remains);
omp_set_lock(&term_lock);
```

If you recall the discussion of busy-waiting in Section 4.5 and Exercise 4.3 of Chapter 4, you may be concerned about the possibility that the compiler might reorder the code around the busy-wait loop. The compiler should not reorder across calls to `omp_set_lock` or `omp_unset_lock`. However, the updates to the variables *could* be reordered, so if we're going to be using compiler optimization, we should declare both with the `volatile` keyword.

Emulating the condition broadcast is straightforward: When a thread determines that there's no work left (Line 14 in Program 6.8), then the condition broadcast (Line 17) can be replaced with the assignment

```
work_remains = 0;  /* Assign false to work_remains */
```

The "awakened" threads can check if they were awakened by some thread's setting `work_remains` to false, and, if they were, return from `Terminated` with the value true.

Emulating the condition signal requires a little more work. The thread that has split its stack needs to choose one of the sleeping threads and set the variable `awakened_thread` to the chosen thread's rank. Thus, at a minimum, we need to keep a list of the ranks of the sleeping threads. A simple way to do this is to use a shared queue of thread ranks. When a thread runs out of work, it enqueues its rank before entering the busy-wait loop. When a thread splits its stack, it can choose the thread to awaken by dequeuing the queue of waiting threads:

```
got_lock = omp_test_lock(&term_lock);
if (got_lock != 0) {
   if (waiting_threads > 0 && new_stack == NULL) {
      Split my_stack creating new_stack;
      awakened_thread = Dequeue(term_queue);
   }
   omp_unset_lock(&term_lock);
}
```

The awakened thread needs to reset `awakened_thread` to -1 before it returns from its call to the `Terminated` function.

Note that there is no danger that some other thread will be awakened before the awakened thread reacquires the lock. As long as `new_stack` is not `NULL`, no thread will attempt to split its stack, and hence no thread will try to awaken another thread. So if several threads call `Terminated` before the awakened thread reacquires the lock, they'll either return if their stacks are nonempty, or they'll enter the wait if their stacks are empty.

6.2.10 Performance of the OpenMP implementations

Table 6.9 shows run-times of the two OpenMP implementations on the same two fifteen-city problems that we used to test the Pthreads implementations. The programs

Table 6.9 Performance of OpenMP and Pthreads Implementations of Tree Search (times in seconds)

	First Problem						Second Problem					
	Static		*Dynamic*				*Static*		*Dynamic*			
Th	*OMP*	*Pth*	*OMP*		*Pth*		*OMP*	*Pth*	*OMP*		*Pth*	
1	32.5	32.7	33.7	(0)	34.7	(0)	25.6	25.8	26.6	(0)	27.5	(0)
2	27.7	27.9	28.0	(6)	28.9	(7)	25.6	25.8	18.8	(9)	19.2	(6)
4	25.4	25.7	33.1	(75)	25.9	(47)	25.6	25.8	9.8	(52)	9.3	(49)
8	28.0	23.8	19.2	(134)	22.4	(180)	23.8	24.0	6.3	(163)	5.7	(256)

were also run on the same system we used for the Pthreads and serial tests. For ease of comparison, we also show the Pthreads run-times. Run-times are in seconds and the numbers in parentheses show the total number of times stacks were split in the dynamic implementations.

For the most part, the OpenMP implementations are comparable to the Pthreads implementations. This isn't surprising since the system on which the programs were run has eight cores, and we wouldn't expect busy-waiting to degrade overall performance unless we were using more threads than cores.

There are two notable exceptions for the first problem. The performance of the static OpenMP implementation with eight threads is much worse than the Pthreads implementation, and the dynamic implementation with four threads is much worse than the Pthreads implementation. This could be a result of the nondeterminism of the programs, but more detailed profiling will be necessary to determine the cause with any certainty.

6.2.11 Implementation of tree search using MPI and static partitioning

The vast majority of the code used in the static parallelizations of tree search using Pthreads and OpenMP is taken straight from the second implementation of serial, iterative tree search. In fact, the only differences are in starting the threads, the initial partitioning of the tree, and the `Update_best_tour` function. We might therefore expect that an MPI implementation would also require relatively few changes to the serial code, and this is, in fact, the case.

There is the usual problem of distributing the input data and collecting the results. In order to construct a complete tour, a process will need to choose an edge into each vertex and out of each vertex. Thus, each tour will require an entry from each row and each column for each city that's added to the tour, so it would clearly be advantageous for each process to have access to the entire adjacency matrix. Note that the adjacency matrix is going to be relatively small. For example, even if we have 100 cities, it's unlikely that the matrix will require more than 80,000 bytes of

storage, so it makes sense to simply read in the matrix on process 0 and broadcast it to all the processes.

Once the processes have copies of the adjacency matrix, the bulk of the tree search can proceed as it did in the Pthreads and OpenMP implementations. The principal differences lie in

- partitioning the tree,
- checking and updating the best tour, and
- after the search has terminated, making sure that process 0 has a copy of the best tour for output.

We'll discuss each of these in turn.

Partitioning the tree

In the Pthreads and OpenMP implementations, thread 0 uses breadth-first search to search the tree until there are at least `thread_count` partial tours. Each thread then determines which of these initial partial tours it should get and pushes its tours onto its local stack. Certainly MPI process 0 can also generate a list of `comm_sz` partial tours. However, since memory isn't shared, it will need to send the initial partial tours to the appropriate process. We could do this using a loop of sends, but distributing the initial partial tours looks an awful lot like a call to `MPI_Scatter`. In fact, the only reason we can't use `MPI_Scatter` is that the number of initial partial tours may not be evenly divisible by `comm_sz`. When this happens, process 0 won't be sending the same number of tours to each process, and `MPI_Scatter` requires that the source of the scatter send the same number of objects to each process in the communicator.

Fortunately, there is a variant of `MPI_Scatter`, `MPI_Scatterv`, which *can* be used to send different numbers of objects to different processes. First recall the syntax of `MPI_Scatter`:

```
int MPI_Scatter(
      void          sendbuf     /* in  */,
      int           sendcount   /* in  */,
      MPI_Datatype  sendtype    /* in  */,
      void*         recvbuf     /* out */,
      int           recvcount   /* in  */,
      MPI_Datatype  recvtype    /* in  */,
      int           root        /* in  */,
      MPI_Comm      comm        /* in  */);
```

Process `root` sends `sendcount` objects of type `sendtype` from `sendbuf` to each process in `comm`. Each process in `comm` receives `recvcount` objects of type `recvtype` into `recvbuf`. Most of the time, `sendtype` and `recvtype` are the same and `sendcount` and `recvcount` are also the same. In any case, it's clear that the `root` process must send the same number of objects to each process.

`MPI_Scatterv`, on the other hand, has syntax

```
int MPI_Scatterv(
      void*        sendbuf         /* in  */,
```

```
      int*          sendcounts      /* in  */,
      int*          displacements   /* in  */,
      MPI_Datatype  sendtype        /* in  */,
      void*         recvbuf         /* out */,
      int           recvcount       /* in  */,
      MPI_Datatype  recvtype        /* in  */,
      int           root            /* in  */,
      MPI_Comm      comm            /* in  */);
```

The single `sendcount` argument in a call to `MPI_Scatter` is replaced by two array arguments: `sendcounts` and `displacements`. Both of these arrays contain `comm_sz` elements: `sendcounts[q]` is the number of objects of type `sendtype` being sent to process q. Furthermore, `displacements[q]` specifies the start of the block that is being sent to process q. The displacement is calculated in units of type `sendtype`. So, for example, if `sendtype` is `MPI_INT`, and `sendbuf` has type `int*`, then the data that is sent to process q will begin in location

```
sendbuf + displacements[q]
```

In general, `displacements[q]` specifies the offset into `sendbuf` of the data that will go to process q. The "units" are measured in blocks with extent equal to the extent of `sendtype`.

Similarly, `MPI_Gatherv` generalizes `MPI_Gather`:

```
int MPI_Gatherv(
      void*         sendbuf          /* in  */,
      int           sendcount        /* in  */,
      MPI_Datatype  sendtype         /* in  */,
      void*         recvbuf          /* out */,
      int*          recvcounts       /* in  */,
      int*          displacements    /* in  */,
      MPI_Datatype  recvtype         /* in  */,
      int           root             /* in  */,
      MPI_Comm      comm             /* in  */);
```

Maintaining the best tour

As we observed in our earlier discussion of parallelizing tree search, having each process use its own best tour is likely to result in a lot of wasted computation since the best tour on one process may be much more costly than most of the tours on another process (see Exercise 6.21). Therefore, when a process finds a new best tour, it should send it to the other processes.

First note that when a process finds a new best tour, it really only needs to send its *cost* to the other processes. Each process only makes use of the cost of the current best tour when it calls `Best_tour`. Also, when a process updates the best tour, it doesn't care what the actual cities on the former best tour were; it only cares that the cost of the former best tour is greater than the cost of the new best tour.

During the tree search, when one process wants to communicate a new best cost to the other processes, it's important to recognize that we can't use `MPI_Bcast`,

for recall that MPI_Bcast is blocking and every process in the communicator must call MPI_Bcast. However, in parallel tree search the only process that will know that a broadcast should be executed is the process that has found a new best cost. If it tries to use MPI_Bcast, it will probably block in the call and never return, since it will be the only process that calls it. We therefore need to arrange that the new tour is sent in such a way that the sending process won't block indefinitely.

MPI provides several options. The simplest is to have the process that finds a new best cost use MPI_Send to send it to all the other processes:

```
for (dest = 0; dest < comm_sz; dest++)
   if (dest != my_rank)
      MPI_Send(&new_best_cost, 1, MPI_INT, dest, NEW_COST_TAG,
          comm);
```

Here, we're using a special tag defined in our program, NEW_COST_TAG. This will tell the receiving process that the message is a new cost–as opposed to some other type of message–for example, a tour.

The destination processes can periodically check for the arrival of new best tour costs. We can't use MPI_Recv to check for messages since it's blocking; if a process calls

```
MPI_Recv(&received_cost, 1, MPI_INT, MPI_ANY_SOURCE, NEW_COST_TAG,
      comm, &status);
```

the process will block until a matching message arrives. If no message arrives—for example, if no process finds a new best cost—the process will hang. Fortunately, MPI provides a function that only *checks* to see if a message is available; it doesn't actually try to receive a message. It's called MPI_Iprobe, and its syntax is

```
int MPI_Iprobe(
            int          source        /* in  */,
            int          tag           /* in  */,
            MPI_Comm     comm          /* in  */,
            int*         msg_avail_p   /* out */,
            MPI_Status*  status_p      /* out */);
```

It checks to see if a message from process rank source in communicator comm and with tag tag is available. If such a message is available, *msg_avail_p will be assigned the value true and the members of *status_p will be assigned the appropriate values. For example, status_p->MPI_SOURCE will be assigned the rank of the source of the message that's been received. If no message is available, *msg_avail_p will be assigned the value false. The source and tag arguments can be the wildcards MPI_ANY_SOURCE and MPI_ANY_TAG, respectively. So, to check for a message with a new cost from any process, we can call

```
MPI_Iprobe(MPI_ANY_SOURCE, NEW_COST_TAG, comm, &msg_avail, &status);
```

```
MPI_Iprobe(MPI_ANY_SOURCE, NEW_COST_TAG, comm, &msg_avail,
      &status);
while (msg_avail) {
   MPI_Recv(&received_cost, 1, MPI_INT, status.MPI_SOURCE,
         NEW_COST_TAG, comm, MPI_STATUS_IGNORE);
   if (received_cost < best_tour_cost)
      best_tour_cost = received_cost;
   MPI_Iprobe(MPI_ANY_SOURCE, NEW_COST_TAG, comm, &msg_avail,
     &status);
}  /* while */
```

Program 6.9: MPI code to check for new best tour costs

If `msg_avail` is true, then we can receive the new cost with a call to `MPI_Recv`:

```
MPI_Recv(&received_cost, 1, MPI_INT, status.MPI_SOURCE,
      NEW_COST_TAG, comm, MPI_STATUS_IGNORE);
```

A natural place to do this is in the `Best_tour` function. Before checking whether our new tour is the best tour, we can check for new tour costs from other processes with the code in Program 6.9.

This code will continue to receive messages with new costs as long as they're available. Each time a new cost is received that's better than the current best cost, the variable `best_tour_cost` will be updated.

Did you spot the potential problem with this scheme? If there is no buffering available for the sender, then the loop of calls to `MPI_Send` can cause the sending process to block until a matching receive is posted. If all the other processes have completed their searches, the sending process will hang. The loop of calls to `MPI_Send` is therefore unsafe.

There are a couple of alternatives provided by MPI: **buffered sends** and **nonblocking sends**. We'll discuss buffered sends here. See Exercise 6.22 for a discussion of nonblocking operations in MPI.

Modes and Buffered Sends

MPI provides four **modes** for sends: **standard**, **synchronous**, **ready**, and **buffered**. The various modes specify different semantics for the sending functions. The send that we first learned about, `MPI_Send`, is the standard mode send. With it, the MPI implementation can decide whether to copy the contents of the message into its own storage or to block until a matching receive is posted. Recall that in synchronous mode, the send will block until a matching receive is posted. In ready mode, the send is erroneous unless a matching receive is posted *before* the send is started. In buffered mode, the MPI implementation must copy the message into local temporary storage if a matching receive hasn't been posted. The local temporary storage must be provided by the user program, not the MPI implementation.

Each mode has a different function: MPI_Send, MPI_Ssend, MPI_Rsend, and MPI_Bsend, respectively, but the argument lists are identical to the argument lists for MPI_Send:

```
int MPI_Xsend(
      void*         message       /* in */,
      int           message_size  /* in */,
      MPI_Datatype  message_type  /* in */,
      int           dest          /* in */,
      int           tag           /* in */,
      MPI_Comm      comm          /* in */);
```

The buffer that's used by MPI_Bsend must be turned over to the MPI implementation with a call to MPI_Buffer_attach:

```
int MPI_Buffer_attach(
      void*  buffer       /* in */,
      int    buffer_size  /* in */);
```

The buffer argument is a pointer to a block of memory allocated by the user program and buffer_size is its size in bytes. A previously "attached" buffer can be reclaimed by the program with a call to

```
int MPI_Buffer_detach(
      void*   buf_p       /* out */,
      int*    buf_size_p  /* out */);
```

The *buf_p argument returns the address of the block of memory that was previously attached, and *buf_size_p gives its size in bytes. A call to MPI_Buffer_detach will block until all messages that have been stored in the buffer are transmitted. Note that since buf_p is an output argument, it should probably be passed in with the ampersand operator. For example:

```
char buffer[1000];
char* buf;
int buf_size;
. . .
MPI_Buffer_attach(buffer, 1000);
. . .
/* Calls to MPI_Bsend */
. . .
MPI_Buffer_detach(&buf, &buf_size);
```

At any point in the program only one user-provided buffer can be attached, so if there may be multiple buffered sends that haven't been completed, we need to estimate the amount of data that will be buffered. Of course, we can't know this with any certainty, but we do know that in any "broadcast" of a best tour, the process doing the broadcast will make comm_sz − 1 calls to MPI_Bsend, and each of these calls will send a single int. We can thus determine the size of the buffer needed for a single broadcast. The amount of storage that's needed for the *data* that's transmitted can be determined with a call to MPI_Pack_size:

```
int MPI_Pack_size(
      int           count     /* in  */,
      MPI_Datatype  datatype  /* in  */,
      MPI_Comm      comm      /* in  */,
      int*          size_p    /* out */);
```

The output argument gives an upper bound on the number of bytes needed to store the data in a message. This won't be enough, however. Recall that in addition to the data, a message stores information such as the destination, the tag, and the communicator, so for each message there is some additional overhead. An upper bound on this additional overhead is given by the MPI constant `MPI_BSEND_OVERHEAD`. For a single broadcast, the following code determines the amount of storage needed:

```
int data_size;
int message_size;
int bcast_buf_size;

MPI_Pack_size(1, MPI_INT, comm, &data_size);
message_size = data_size + MPI_BSEND_OVERHEAD;
bcast_buf_size = (comm_sz - 1)*message_size;
```

We should guess a generous upper bound on the number of broadcasts and multiply that by `bcast_buf_size` to get the size of the buffer to attach.

Printing the best tour

When the program finishes, we'll want to print out the actual tour as well as its cost, so we do need to get the tour to process 0. It might at first seem that we could arrange this by having each process store its local best tour—the best tour that it finds—and when the tree search has completed, each process can check its local best tour cost and compare it to the global best tour cost. If they're the same, the process could send its local best tour to process 0. There are, however, several problems with this. First, it's entirely possible that there are multiple "best" tours in the TSP digraph, tours that all have the same cost, and different processes may find these different tours. If this happens, multiple processes will try to send their best tours to process 0, and all but one of the threads could hang in a call to `MPI_Send`. A second problem is that it's possible that one or more processes never received the best tour cost, and they may try to send a tour that isn't optimal.

We can avoid these problems by having each process store its local best tour, but after all the processes have completed their searches, they can all call `MPI_Allreduce` and the process with the global best tour can then send it to process 0 for output. The following pseudocode provides some details:

```
struct {
   int cost;
   int rank;
} loc_data, global_data;

loc_data.cost = Tour_cost(loc_best_tour);
loc_data.rank = my_rank;
```

```
MPI_Allreduce(&loc_data, &global_data, 1, MPI_2INT, MPI_MINLOC,
      comm);
if (global_data.rank == 0) return;
      /* 0 already has the best tour */
if (my_rank == 0)
   Receive best tour from process global_data.rank;
else if (my_rank == global_data.rank)
   Send best tour to process 0;
```

The key here is the operation we use in the call to `MPI_Allreduce`. If we just used `MPI_MIN`, we would know what the cost of the global best tour was, but we wouldn't know who owned it. However, MPI provides a predefined operator, `MPI_MINLOC`, which operates on pairs of values. The first value is the value to be minimized—in our setting, the cost of the tour—and the second value is the *location* of the minimum—in our setting, the rank of the process that actually owns the best tour. If more than one process owns a tour with minimum cost, the location will be the lowest of the ranks of the processes that own a minimum cost tour. The input and the output buffers in the call to `MPI_Allreduce` are two-member structs. Since both the cost and the rank are `int`s, both members are `int`s. Note that MPI also provides a predefined type `MPI_2INT` for this type. When the call to `MPI_Allreduce` returns, we have two alternatives:

- If process 0 already has the best tour, we simply return.
- Otherwise, the process owning the best tour sends it to process 0.

Unreceived messages

As we noted in the preceding discussion, it is possible that some messages won't be received during the execution of the parallel tree search. A process may finish searching its subtree before some other process has found a best tour. This won't cause the program to print an incorrect result; the call to `MPI_Allreduce` that finds the process with the best tour won't return until every process has called it, and some process will have the best tour. Thus, it will return with the correct least-cost tour, and process 0 will receive this tour.

However, unreceived messages can cause problems with the call to `MPI_Buffer_detach` or the call to `MPI_Finalize`. A process can hang in one of these calls if it is storing buffered messages that were never received, so before we attempt to shut down MPI, we can try to receive any outstanding messages by using `MPI_Iprobe`. The code is very similar to the code we used to check for new best tour costs. See Program 6.9. In fact, the only messages that are not sent in collectives are the "best tour" message sent to process 0, and the best tour cost broadcasts. The MPI collectives will hang if some process doesn't participate, so we only need to look for unreceived best tours.

In the dynamically load-balanced code (which we'll discuss shortly) there are other messages, including some that are potentially quite large. To handle this situation, we can use the `status` argument returned by `MPI_Iprobe` to determine the size of the message and allocate additional storage as necessary (see Exercise 6.23).

6.2.12 Implementation of tree search using MPI and dynamic partitioning

In an MPI program that dynamically partitions the search tree, we can try to emulate the dynamic partitioning that we used in the Pthreads and OpenMP programs. Recall that in those programs, before each pass through the main `while` loop in the search function, a thread called a boolean-valued function called `Terminated`. When a thread ran out of work—that is, its stack was empty—it went into a condition wait (Pthreads) or a busy-wait (OpenMP) until it either received additional work or it was notified that there was no more work. In the first case, it returned to searching for a best tour. In the second case, it quit. A thread that had at least two records on its stack would give half of its stack to one of the waiting threads.

Much of this can be emulated in a distributed-memory setting. When a process runs out of work, there's no condition wait, but it can enter a busy-wait, in which it waits to either receive more work or notification that the program is terminating. Similarly, a process with work can split its stack and send work to an idle process.

The key difference is that there is no central repository of information on which processes are waiting for work, so a process that splits its stack can't just dequeue a queue of waiting processes or call a function such as `pthread_cond_signal`. It needs to "know" a process that's waiting for work so it can send the waiting process more work. Thus, rather than simply going into a busy-wait for additional work or termination, a process that has run out of work should send a request for work to another process. If it does this, then, when a process enters the `Terminated` function, it can check to see if there's a request for work from some other process. If there is, and the process that has just entered `Terminated` has work, it can send part of its stack to the requesting process. If there is a request, and the process has no work available, it can send a rejection. Thus, when we have distributed-memory, pseudocode for our `Terminated` function can look something like the pseudocode shown in Program 6.10.

`Terminated` begins by checking on the number of tours that the process has in its stack (Line 1); if it has at least two that are "worth sending," it calls `Fulfill_request` (Line 2). `Fulfill_request` checks to see if the process has received a request for work. If it has, it splits its stack and sends work to the requesting process. If it hasn't received a request, it just returns. In either case, when it returns from `Fulfill_request` it returns from `Terminated` and continues searching.

If the calling process doesn't have at least two tours worth sending, `Terminated` calls `Send_rejects` (Line 5), which checks for any work requests from other processes and sends a "no work" reply to each requesting process. After this, `Terminated` checks to see if the calling process has any work at all. If it does—that is, if its stack isn't empty—it returns and continues searching.

Things get interesting when the calling process has no work left (Line 9). If there's only one process in the communicator (`comm_sz` = 1), then the process returns from `Terminated` and quits. If there's more than one process, then the process "announces" that it's out of work in Line 11. This is part of the implementation

```
1  if (My_avail_tour_count(my_stack) >= 2) {
2     Fulfill_request(my_stack);
3     return false;  /* Still more work */
4  } else { /* At most 1 available tour */
5     Send_rejects();  /* Tell everyone who's requested */
6                      /* work that I have none         */
7     if (!Empty_stack(my_stack)) {
8        return false;  /* Still more work */
9     } else {  /* Empty stack */
10       if (comm_sz == 1) return true;
11       Out_of_work();
12       work_request_sent = false;
13       while (1) {
14          Clear_msgs();  /* Msgs unrelated to work, termination */
15          if (No_work_left()) {
16             return true;  /* No work left.  Quit */
17          } else if (!work_request_sent) {
18             Send_work_request();  /* Request work from someone */
19             work_request_sent = true;
20          } else {
21             Check_for_work(&work_request_sent, &work_avail);
22             if (work_avail) {
23                Receive_work(my_stack);
24                return false;
25             }
26          }
27       }  /* while */
28    } /* Empty stack */
29 } /* At most 1 available tour */
```

Program 6.10: `Terminated` function for a dynamically partitioned TSP solver that uses MPI

of a "distributed termination detection algorithm," which we'll discuss shortly. For now, let's just note that the termination detection algorithm that we used with shared-memory may not work, since it's impossible to guarantee that a variable storing the number of processes that have run out of work is up to date.

Before entering the apparently infinite `while` loop (Line 13), we set the variable `work_request_sent` to false (Line 12). As its name suggests, this variable tells us whether we've sent a request for work to another process; if we have, we know that we should wait for work or a message saying "no work available" from that process before sending out a request to another process.

The `while(1)` loop is the distributed-memory version of the OpenMP busy-wait loop. We are essentially waiting until we either receive work from another process or we receive word that the search has been completed.

When we enter the `while(1)` loop, we deal with any outstanding messages in Line 14. We may have received updates to the best tour cost and we may have received requests for work. It's essential that we tell processes that have requested

work that we have none, so that they don't wait forever when there's no work available. It's also a good idea to deal with updates to the best tour cost, since this will free up space in the sending process' message buffer.

After clearing out outstanding messages, we iterate through the possibilities:

- The search has been completed, in which case we quit (Lines 15–16).
- We don't have an outstanding request for work, so we choose a process and send it a request (Lines 17–19). We'll take a closer look at the problem of which process should be sent a request shortly.
- We do have an outstanding request for work (Lines 21–25). So we check whether the request has been fulfilled or rejected. If it has been fulfilled, we receive the new work and return to searching. If we received a rejection, we set `work_request_sent` to false and continue in the loop. If the request was neither fulfilled nor rejected, we also continue in the `while(1)` loop.

Let's take a closer look at some of these functions.

`My_avail_tour_count`

The function `My_avail_tour_count` can simply return the size of the process' stack. It can also make use of a "cutoff length." When a partial tour has already visited most of the cities, there will be very little work associated with the subtree rooted at the partial tour. Since sending a partial tour is likely to be a relatively expensive operation, it may make sense to only send partial tours with fewer than some cutoff number of edges. In Exercise 6.24 we take a look at how such a cutoff affects the overall run-time of the program.

`Fulfill_request`

If a process has enough work so that it can usefully split its stack, it calls `Fulfill_request` (Line 2). `Fulfill_request` uses `MPI_Iprobe` to check for a request for work from another process. If there is a request, it receives it, splits its stack, and sends work to the requesting process. If there isn't a request for work, the process just returns.

Splitting the stack

A `Split_stack` function is called by `Fulfill_request`. It uses the same basic algorithm as the Pthreads and OpenMP functions, that is, alternate partial tours with fewer than `split_cutoff` cities are collected for sending to the process that has requested work. However, in the shared-memory programs, we simply copy the tours (which are pointers) from the original stack to a new stack. Unfortunately, because of the pointers involved in the new stack, such a data structure cannot be simply sent to another process (see Exercise 6.25). Thus, the MPI version of `Split_stack` *packs* the contents of the new stack into contiguous memory and sends the block of contiguous memory, which is *unpacked* by the receiver into a new stack.

MPI provides a function, `MPI_Pack`, for packing data into a buffer of contiguous memory. It also provides a function, `MPI_Unpack`, for unpacking data from a buffer

of contiguous memory. We took a brief look at them in Exercise 6.20 of Chapter 3. Recall that their syntax is

```
int MPI_Pack(
      void*         data_to_be_packed    /* in     */,
      int           to_be_packed_count   /* in     */,
      MPI_Datatype  datatype             /* in     */,
      void*         contig_buf           /* out    */,
      int           contig_buf_size      /* in     */,
      int*          position_p           /* in/out */,
      MPI_Comm      comm                 /* in     */);

int MPI_Unpack(
      void*         contig_buf           /* in     */,
      int           contig_buf_size      /* in     */,
      int*          position_p           /* in/out */,
      void*         unpacked_data        /* out    */,
      int           unpack_count         /* in     */,
      MPI_Datatype  datatype             /* in     */,
      MPI_Comm      comm                 /* in     */);
```

`MPI_Pack` takes the data in `data_to_be_packed` and packs it into `contig_buf`. The `*position_p` argument keeps track of where we are in `contig_buf`. When the function is called, it should refer to the first available location in `contig_buf` before `data_to_be_packed` is added. When the function returns, it should refer to the first available location in `contig_buf` after `data_to_be_packed` has been added.

`MPI_Unpack` reverses the process. It takes the data in `contig_buf` and unpacks it into `unpacked_data`. When the function is called, `*position_p` should refer to the first location in `contig_buf` that hasn't been unpacked. When it returns, `*position_p` should refer to the next location in `contig_buf` after the data that was just unpacked.

As an example, suppose that a program contains the following definitions:

```
typedef struct {
   int* cities; /* Cities in partial tour           */
   int count;   /* Number of cities in partial tour */
   int cost;    /* Cost of partial tour             */
} tour_struct;
typedef tour_struct* tour_t;
```

Then we can send a variable with type `tour_t` using the following code:

```
void Send_tour(tour_t tour, int dest) {
   int position = 0;

   MPI_Pack(tour->cities, n+1, MPI_INT, contig_buf, LARGE,
         &position, comm);
   MPI_Pack(&tour->count, 1, MPI_INT, contig_buf, LARGE,
         &position, comm);
   MPI_Pack(&tour->cost, 1, MPI_INT, contig_buf, LARGE,
```

```
            &position, comm);
      MPI_Send(contig_buf, position, MPI_PACKED, dest, 0, comm);
   }  /* Send_tour */
```

Similarly, we can receive a variable of type `tour_t` using the following code:

```
void Receive_tour(tour_t tour, int src) {
   int position = 0;

   MPI_Recv(contig_buf, LARGE, MPI_PACKED, src, 0, comm,
         MPI_STATUS_IGNORE);
   MPI_Unpack(contig_buf, LARGE, &position, tour->cities, n+1,
         MPI_INT, comm);
   MPI_Unpack(contig_buf, LARGE, &position, &tour->count, 1,
         MPI_INT, comm);
   MPI_Unpack(contig_buf, LARGE, &position, &tour->cost, 1,
         MPI_INT, comm);
}  /* Receive_tour */
```

Note that the MPI datatype that we use for sending and receiving packed buffers is `MPI_PACKED`.

Send_rejects

The `Send_rejects` function (Line 5) is similar to the function that looks for new best tours. It uses `MPI_Iprobe` to search for messages that have requested work. Such messages can be identified by a special tag value, for example, `WORK_REQ_TAG`. When such a message is found, it's received, and a reply is sent indicating that there is no work available. Note that both the request for work and the reply indicating there is no work can be messages with zero elements, since the tag alone informs the receiver of the message's purpose. Even though such messages have no content outside of the envelope, the envelope does take space and they need to be received.

Distributed termination detection

The functions `Out_of_work` and `No_work_left` (Lines 11 and 15) implement the termination detection algorithm. As we noted earlier, an algorithm that's modeled on the termination detection algorithm we used in the shared-memory programs will have problems. To see this, suppose each process stores a variable `oow`, which stores the number of processes that are out of work. The variable is set to 0 when the program starts. Each time a process runs out of work, it sends a message to all the other processes saying it's out of work so that all the processes will increment their copies of `oow`. Similarly, when a process receives work from another process, it sends a message to every process informing them of this, and each process will decrement its copy of `oow`. Now suppose we have three process, and process 2 has work but processes 0 and 1 have run out of work. Consider the sequence of events shown in Table 6.10.

The error here is that the work sent from process 1 to process 0 is lost. The reason is that process 0 receives the notification that process 2 is out of work before it receives the notification that process 1 has received work. This may seem improbable,

Table 6.10 Termination Events that Result in an Error

Time	Process 0	Process 1	Process 2
0	Out of Work Notify 1, 2 oow = 1	Out of Work Notify 0, 2 oow = 1	Working oow = 0
1	Send request to 1 oow = 1	Send Request to 2 oow = 1	Recv notify fr 1 oow = 1
2	oow = 1	Recv notify fr 0 oow = 2	Recv request fr 1 oow = 1
3	oow = 1	oow = 2	Send work to 1 oow = 0
4	oow = 1	Recv work fr 2 oow = 1	Recv notify fr 0 oow = 1
5	oow = 1	Notify 0 oow = 1	Working oow = 1
6	oow = 1	Recv request fr 0 oow = 1	Out of work Notify 0, 1 oow = 2
7	Recv notify fr 2 oow = 2	Send work to 0 oow = 0	Send request to 1 oow = 2
8	Recv 1st notify fr 1 oow = 3	Recv notify fr 2 oow = 1	oow = 2
9	Quit	Recv request fr 2 oow = 1	oow = 2

but it's not improbable that process 1 was, for example, interrupted by the operating system and its message wasn't transmitted until after the message from process 2 was transmitted.

Although MPI guarantees that two messages sent from process A to process B will, in general, be received in the order in which they were sent, it makes no guarantee about the order in which messages will be received if they were sent by different processes. This is perfectly reasonable in light of the fact that different processes will, for various reasons, work at different speeds.

Distributed termination detection is a challenging problem, and much work has gone into developing algorithms that are guaranteed to correctly detect it. Conceptually, the simplest of these algorithms relies on keeping track of a quantity that is conserved and can be measured precisely. Let's call it *energy,* since, of course, energy is conserved. At the start of the program, each process has 1 unit of energy. When a process runs out of work, it sends its energy to process 0. When a process fulfills a request for work, it divides its energy in half, keeping half for itself, and sending half to the process that's receiving the work. Since energy is conserved and since the program started with `comm_sz` units, the program should terminate when process 0 finds that it has received a total of `comm_sz` units.

The `Out_of_work` function when executed by a process other than 0 sends its energy to process 0. Process 0 can just add its energy to a `received_energy` variable. The `No_work_left` function also depends on whether process 0 or some other process is calling. If process 0 is calling, it can receive any outstanding messages sent by `Out_of_work` and add the energy into `received_energy`. If `received_energy` equals `comm_sz`, process 0 can send a termination message (with a special tag) to every process. On the other hand, a process other than 0 can just check to see if there's a message with the termination tag.

The tricky part here is making sure that no energy is inadvertently lost; if we try to use `floats` or `doubles`, we'll almost certainly run into trouble since at some point dividing by two will result in underflow. Since the amount of energy in exact arithmetic can be represented by a common fraction, we can represent the amount of energy on each process exactly by a pair of fixed-point numbers. The denominator will always be a power of two, so we can represent it by its base-two logarithm. For a large problem it is possible that the numerator could overflow. However, if this becomes a problem, there are libraries that provide arbitrary precision rational numbers (e.g, GMP [21]). An alternate solution is explored in Exercise 6.26.

Sending requests for work

Once we've decided on which process we plan to send a request to, we can just send a zero-length message with a "request for work" tag. However, there are many possibilities for choosing a destination:

1. Loop through the processes in round-robin fashion. Start with `(my_rank + 1) % comm_sz` and increment this destination (modulo `comm_sz`) each time a new request is made. A potential problem here is that two processes can get "in synch" and request work from the same destination repeatedly.
2. Keep a global destination for requests on process 0. When a process runs out of work, it first requests the current value of the global destination from 0. Process 0 can increment this value (modulo `comm_sz`) each time there's a request. This avoids the issue of multiple processes requesting work from the same destination, but clearly process 0 can become a bottleneck.
3. Each process uses a random number generator to generate destinations. While it can still happen that several processes may simultaneously request work from the same process, the random choice of successive process ranks should reduce the chance that several processes will make repeated requests to the same process.

These are three possible options. We'll explore these options in Exercise 6.29. Also see [22] for an analysis of the options.

Checking for and receiving work

Once a request is sent for work, it's critical that the sending process repeatedly check for a response from the destination. In fact, a subtle point here is that it's critical that the sending process check for a message from the destination process with a "work available tag" or a "no work available tag." If the sending process simply checks

for a message from the destination, it may be "distracted" by other messages from the destination and never receive work that's been sent. For example, there might be a message from the destination requesting work that would mask the presence of a message containing work.

The `Check_for_work` function should therefore first probe for a message from the destination indicating work is available, and, if there isn't such a message, it should probe for a message from the destination saying there's no work available. If there is work available, the `Receive_work` function can receive the message with work and unpack the contents of the message buffer into the process' stack. Note also that it needs to unpack the energy sent by the destination process.

Performance of the MPI programs

Table 6.11 shows the performance of the two MPI programs on the same two fifteen-city problems on which we tested the Pthreads and the OpenMP implementations. Run-times are in seconds and the numbers in parentheses show the total number of times stacks were split in the dynamic implementations. These results were obtained on a different system from the system on which we obtained the Pthreads results. We've also included the Pthreads results for this system, so that the two sets of results can be compared. The nodes of this system only have four cores, so the Pthreads results don't include times for 8 or 16 threads. The cutoff number of cities for the MPI runs was 12.

The nodes of this system are small shared-memory systems, so communication through shared variables should be much faster than distributed-memory communication, and it's not surprising that in every instance the Pthreads implementation outperforms the MPI implementation.

The cost of stack splitting in the MPI implementation is quite high; in addition to the cost of the communication, the packing and unpacking is very time-consuming. It's also therefore not surprising that for relatively small problems with few processes, the static MPI parallelization outperforms the dynamic parallelization. However, the

Table 6.11 Performance of MPI and Pthreads Implementations of Tree Search (times in seconds)

	First Problem						Second Problem					
	Static		*Dynamic*				*Static*		*Dynamic*			
Th/Pr	*Pth*	*MPI*	*Pth*		*MPI*		*Pth*	*MPI*	*Pth*		*MPI*	
1	35.8	40.9	41.9	(0)	56.5	(0)	27.4	31.5	32.3	(0)	43.8	(0)
2	29.9	34.9	34.3	(9)	55.6	(5)	27.4	31.5	22.0	(8)	37.4	(9)
4	27.2	31.7	30.2	(55)	52.6	(85)	27.4	31.5	10.7	(44)	21.8	(76)
8		35.7			45.5	(165)		35.7			16.5	(161)
16		20.1			10.5	(441)		17.8			0.1	(173)

8- and 16-process results suggest that if a problem is large enough to warrant the use of many processes, the dynamic MPI program is much more scalable, and it can provide far superior performance. This is borne out by examination of a 17-city problem run with 16 processes: the dynamic MPI implementation has a run-time of 296 seconds, while the static implementation has a run-time of 601 seconds.

Note that times such as 0.1 second for the second problem running with 16 processes don't really show superlinear speedup. Rather, the initial distribution of work has allowed one of the processes to find the best tour much faster than the initial distributions with fewer processes, and the dynamic partitioning has allowed the processes to do a much better job of load balancing.

6.3 A WORD OF CAUTION

In developing our solutions to the *n*-body problem and TSP, we chose our serial algorithms because they were easy to understand and their parallelization was relatively straightforward. In no case did we choose a serial algorithm because it was the fastest or because it could solve the largest problem. Thus, it should not be assumed that either the serial or the parallel solutions are the best available. For information on "state-of-the-art" algorithms, see the bibliography, especially [12] for the *n*-body problem and [22] for parallel tree search.

6.4 WHICH API?

How can we decide which API, MPI, Pthreads, or OpenMP is best for our application? In general, there are many factors to consider, and the answer may not be at all clear cut. However, here are a few points to consider.

As a first step, decide whether to use distributed-memory, or shared-memory. In order to do this, first consider the amount of memory the application will need. In general, distributed-memory systems can provide considerably more main memory than shared-memory systems, so if the memory requirements are very large, you may need to write the application using MPI.

If the problem will fit into the main memory of your shared-memory system, you may still want to consider using MPI. Since the total available cache on a distributed-memory system will probably be much greater than that available on a shared-memory system, it's conceivable that a problem that requires lots of main memory accesses on a shared-memory system will mostly access cache on a distributed-memory system, and, consequently, have much better overall performance.

However, even if you'll get a big performance improvement from the large aggregate cache on a distributed-memory system, if you already have a large and complex serial program, it often makes sense to write a shared-memory program. It's often

possible to reuse considerably more serial code in a shared-memory program than a distributed-memory program. It's more likely that the serial data structures can be easily adapted to a shared-memory system. If this is the case, the development effort for the shared-memory program will probably be much less. This is especially true for OpenMP programs, since some serial programs can be parallelized by simply inserting some OpenMP directives.

Another consideration is the communication requirements of the parallel algorithm. If the processes/threads do little communication, an MPI program should be fairly easy to develop, and very scalable. At the other extreme, if the processes/threads need to be very closely coordinated, a distributed-memory program will probably have problems scaling to large numbers of processes, and the performance of a shared-memory program should be better.

If you decided that shared-memory is preferable, you will need to think about the details of parallelizing the program. As we noted earlier, if you already have a large, complex serial program, you should see if it lends itself to OpenMP. For example, if large parts of the program can be parallelized with `parallel for` directives, OpenMP will be much easier to use than Pthreads. On the other hand, if the program involves complex synchronization among the threads—for example, read-write locks or threads waiting on signals from other threads—then Pthreads will be much easier to use.

6.5 SUMMARY

In this chapter, we've looked at serial and parallel solutions to two very different problems: the *n*-body problem and solving the traveling salesperson problem using tree search. In each case we began by studying the problem and looking at serial algorithms for solving the problem. We continued by using Foster's methodology for devising a parallel solution, and then, using the designs developed with Foster's methodology, we implemented parallel solutions using Pthreads, OpenMP, and MPI. In developing the reduced MPI solution to the *n*-body problem, we determined that the "obvious" solution would be extremely difficult to implement correctly and would require a huge amount of communication. We therefore turned to an alternative "ring pass" algorithm, which proved to be much easier to implement and is probably more scalable.

In the dynamically partitioned solutions for parallel tree search, we used different methods for the three APIs. With Pthreads, we used a condition variable both for communicating new work among the threads and for termination. OpenMP doesn't provide an analog to Pthreads condition variables, so we used busy-waiting instead. In MPI, since all data is local, we needed to use a more complicated scheme to redistribute work, in which a process that runs out of work chooses a destination process and requests work from that process. To implement this correctly, a process that runs out of work enters a busy-wait loop in which it requests work, looks for a response to the work request, and looks for a termination message.

We saw that in a distributed-memory environment in which processes send each other work, determining when to terminate is a nontrivial problem. We also looked at a relatively straightforward solution to the problem of distributed termination detection, in which there is a fixed amount of "energy" throughout the execution of the program. When processes run out of work they send their energy to process 0, and when processes send work to other processes, they also send half of their current energy. Thus, when process 0 finds that it has all the energy, there is no more work, and it can send a termination message.

In closing, we looked briefly at the problem of deciding which API to use. The first consideration is whether to use shared-memory or distributed-memory. To decide this, we should look at the memory requirements of the application and the amount of communication among the processes/threads. If the memory requirements are great or the distributed-memory version can work mainly with cache, then a distributed-memory program is likely to be much faster. On the other hand, if there is considerable communication, a shared-memory program will probably be faster.

In choosing between OpenMP and Pthreads, if there's an existing serial program and it can be parallelized by the insertion of OpenMP directives, then OpenMP is probably the clear choice. However, if complex thread synchronization is needed—for example, read-write locks or thread signaling—then Pthreads will be easier to use. In the course of developing these programs, we also learned some more about Pthreads, OpenMP, and MPI.

6.5.1 Pthreads and OpenMP

In tree search, we need to check the cost of the current best tour before updating the best tour. In the Pthreads and OpenMP implementations of parallel tree search, updating the best tour introduces a race condition. A thread that wants to update the best tour must therefore first acquire a lock. The combination of "test lock condition" and "update lock condition" can cause a problem: the lock condition (e.g. the cost of the best tour) can change between the time of the first test and the time that the lock is acquired. Thus, the threads also need to check the lock condition *after* they acquire the lock, so pseudocode for updating the best tour should look something like this:

```
if (new_tour_cost < best_tour_cost) {
   Acquire lock protecting best tour;
   if (new_tour_cost < best_tour_cost)
      Update best tour;
   Relinquish lock;
}
```

Remember that we have also learned that Pthreads has a *nonblocking* version of `pthreads_mutex_lock` called `pthread_mutex_trylock`. This function checks to see if the mutex is available. If it is, it acquires the mutex and returns the value 0. If the mutex isn't available, instead of waiting for it to become available, it will return a nonzero value.

The analog of pthread_mutex_trylock in OpenMP is omp_test_lock. However, its return values are the opposite of those for pthread_mutex_trylock: it returns a nonzero value if the lock is acquired and a zero value if the lock is not acquired.

When a single thread should execute a structured block, OpenMP provides a couple of alternatives to the test, and:

```
if (my_rank == special_rank) {
   Execute action;
}
```

With the single directive

```
#     pragma omp single
      Execute action;

      Next action;
```

the run-time system will choose a single thread to execute the action. The other threads will wait in an implicit barrier before proceeding to Next action. With the master directive

```
#     pragma omp master
      Execute action;

      Next action;
```

the master thread (thread 0) will execute the action. However, unlike the single directive, there is no implicit barrier after the block Execute action, and the other threads in the team will proceed immediately to execute Next action. Of course, if we need a barrier before proceeding, we can add an explicit barrier after completing the structured block Execute action. In Exercise 6.6 we see that OpenMP provides a nowait clause which can modify a single directive:

```
#     pragma omp single nowait
      Execute action;

      Next action;
```

When this clause is added, the thread selected by the run-time system to execute the action will execute it as before. However, the other threads in the team won't wait, they'll proceed immediately to execute Next action. The nowait clause can also be used to modify parallel for and for directives.

6.5.2 MPI

We learned quite a bit more about MPI. We saw that in some of the collective communication functions that use an input and an output buffer, we can use the argument MPI_IN_PLACE so that the input and output buffers are the same. This can save on memory and the implementation may be able to avoid copying from the input buffer to the output buffer.

The functions MPI_Scatter and MPI_Gather can be use to split an array of data among processes and collect distributed data into a single array, respectively. However, they can only be used when the amount of data going to or coming from each process is the same for each process. If we need to assign different amounts of data to each process, or to collect different amounts of data from each process, we can use MPI_Scatterv and MPI_Gatherv, respectively:

```
int MPI_Scatterv(
         void*        sendbuf          /* in  */,
         int*         sendcounts       /* in  */,
         int*         displacements    /* in  */,
         MPI_Datatype sendtype         /* in  */,
         void*        recvbuf          /* out */,
         int          recvcount        /* in  */,
         MPI_Datatype recvtype         /* in  */,
         int          root             /* in  */,
         MPI_Comm     comm             /* in  */);

int MPI_Gatherv(
         void*        sendbuf          /* in  */,
         int          sendcount        /* in  */,
         MPI_Datatype sendtype         /* in  */,
         void*        recvbuf          /* out */,
         int*         recvcounts       /* in  */,
         int*         displacements    /* in  */,
         MPI_Datatype recvtype         /* in  */,
         int          root             /* in  */,
         MPI_Comm     comm             /* in  */);
```

The arguments sendcounts for MPI_Scatterv and recvcounts for MPI_Gatherv are arrays with comm_sz elements. They specify the amount of data (in units of sendtype/recvtype) going to or coming from each process. The displacements arguments are also arrays with comm_sz elements. They specify the offsets (in units of sendtype/recvtype) of the data going to or coming from each process.

We saw that there is a special operator, MPI_MIN_LOC, that can be used in calls to MPI_Reduce and MPI_Allreduce. It operates on pairs of values and returns a pair of values. If the pairs are

$$(a_0, b_0), (a_1, b_1), \ldots, (a_{\texttt{comm\_sz}-1}, b_{\texttt{comm\_sz}-1}),$$

suppose that a is the minimum of the a_i's and q is the smallest process rank at which a occurs. Then the MPI_MIN_LOC operator will return the pair (a_q, b_q). We used this to find not only the cost of the minimum-cost tour, but by making the b_i's the process ranks, we determined which process owned the minimum-cost tour.

In our development of two MPI implementations of parallel tree search, we made repeated use of MPI_Iprobe:

```
int MPI_Iprobe(
          int         source        /* in  */,
          int         tag           /* in  */,
```

```
MPI_Comm    comm          /* in  */,
int*        msg_avail_p   /* out */,
MPI_Status  status_p      /* out */);
```

It checks to see if there is a message from `source` with tag `tag` available to be received. If such a message is available, `msg_avail_p` will be given the value true. Note that `MPI_Iprobe` doesn't actually receive the message, but if such a message is available, a call to `MPI_Recv` will receive it. Both the `source` and `tag` arguments can be the wildcards `MPI_ANY_SOURCE` and `MPI_ANY_TAG`, respectively. For example, we often wanted to check whether any process had sent a message with a new best cost. We checked for the arrival of such a message with the call

```
MPI_Iprobe(MPI_ANY_SOURCE, NEW_COST_TAG, comm, &msg_avail,
      &status);
```

If such a message is available, its source will be returned in the `*status_p` argument. Thus, `status.MPI_SOURCE` can be used to receive the message:

```
MPI_Recv(&new_cost, 1, MPI_INT, status.MPI_SOURCE, NEW_COST_TAG,
      comm, MPI_STATUS_IGNORE);
```

There were several occasions when we wanted a send function to return immediately, regardless of whether the message had actually been sent. One way to arrange this in MPI is to use **buffered send mode**. In a buffered send, the user program provides storage for messages with a call to `MPI_Buffer_attach`. Then when the program sends the message with `MPI_Bsend`, the message is either transmitted immediately or copied to the user-program-provided buffer. In either case the call returns without blocking. When the program no longer needs to use buffered send mode, the buffer can be recovered with a call to `MPI_Buffer_detach`.

We also saw that MPI provides three other modes for sending: **synchronous**, **standard**, and **ready**. Synchronous sends won't buffer the data; a call to the synchronous send function `MPI_Ssend` won't return until the receiver has begun receiving the data. Ready sends (`MPI_Rsend`) are erroneous unless the matching receive has already been started when `MPI_Rsend` is called. The ordinary send `MPI_Send` is called the standard mode send.

In Exercise 6.22 we explore an alternative to buffered mode: nonblocking sends. As the name suggests, a nonblocking send returns regardless of whether the message has been transmitted. However, the send must be completed by calling one of several functions that *wait* for completion of the nonblocking operation. There is also a nonblocking receive function.

Since addresses on one system will, in general, have no relation to addresses on another system, pointers should not be sent in MPI messages. If you're using data structures that have embedded pointers, MPI provides the function `MPI_Pack` for storing a data structure in a single, contiguous buffer before sending. Similarly, the function `MPI_Unpack` can be used to take data that's been received into a single contiguous buffer and unpack it into a local data structure. Their syntax is

```
int MPI_Pack(
      void*         data_to_be_packed    /* in     */,
      int           to_be_packed_count   /* in     */,
      MPI_Datatype  datatype             /* in     */,
      void*         contig_buf           /* out    */,
      int           contig_buf_size      /* in     */,
      int*          position_p           /* in/out */,
      MPI_Comm      comm                 /* in     */);

int MPI_Unpack(
      void*         contig_buf           /* in     */,
      int           contig_buf_size      /* in     */,
      int*          position_p           /* in/out */,
      void*         unpacked_data        /* out    */,
      int           unpack_count         /* in     */,
      MPI_Datatype  datatype             /* in     */,
      MPI_Comm      comm                 /* in     */);
```

The key to their use is the `position_p` argument. When `MPI_Pack` is called, it should reference the first available location in `contig_buf`. So, for example, when we start packing the data `*position_p` should be set to 0. When `MPI_Pack` returns, `*position_p` will refer to the first available location following the data that was just packed. Thus, successive elements of a data structure can be packed into a single buffer by repeated calls to `MPI_Pack`. When a packed buffer is received, the data can be unpacked in a completely analogous fashion. Note that when a buffer packed with `MPI_Pack` is sent, the datatype for both the send and the receive should be `MPI_PACKED`.

6.6 EXERCISES

6.1. In each iteration of the serial *n*-body solver, we first compute the total force on each particle, and then we compute the position and velocity of each particle. Would it be possible to reorganize the calculations so that in each iteration we did all of the calculations for each particle before proceeding to the next particle? That is, could we use the following pseudocode?

```
for each timestep
   for each particle {
      Compute total force on particle;
      Find position and velocity of particle;
      Print position and velocity of particle;
   }
```

If so, what other modifications would we need to make to the solver? If not, why not?

6.2. Run the basic serial *n*-body solver for 1000 timesteps with a stepsize of 0.05, no output, and internally generated initial conditions. Let the number

of particles range from 500 to 2000. How does the run-time change as the number of particles increases? Can you extrapolate and predict how many particles the solver could handle if it ran for 24 hours?

6.3. Parallelize the reduced version of the *n*-body solver with OpenMP or Pthreads and a single `critical` directive (OpenMP) or a single mutex (Pthreads) to protect access to the `forces` array. Parallelize the rest of the solver by parallelizing the inner `for` loops. How does the performance of this code compare with the performance of the serial solver? Explain your answer.

6.4. Parallelize the reduced version of the *n*-body solver with OpenMP or Pthreads and a lock/mutex for each particle. The locks/mutexes should be used to protect access to updates to the `forces` array. Parallelize the rest of the solver by parallelizing the inner `for` loops. How does the performance compare with the performance of the serial solver? Explain your answer.

6.5. In the shared-memory reduced *n*-body solver, if we use a block partition in both phases of the calculation of the forces, the loop in the second phase can be changed so that the `for thread` loop only goes up to `my_rank` instead of `thread_count`. That is, the code

```
#       pragma omp for
        for (part = 0; part < n; part++) {
           forces[part][X] = forces[part][Y] = 0.0;
           for (thread = 0; thread < thread_count; thread++) {
              forces[part][X] += loc_forces[thread][part][X];
              forces[part][Y] += loc_forces[thread][part][Y];
           }
        }
```

can be changed to

```
#       pragma omp for
        for (part = 0; part < n; part++) {
           forces[part][X] = forces[part][Y] = 0.0;
           for (thread = 0; thread < my_rank; thread++) {
              forces[part][X] += loc_forces[thread][part][X];
              forces[part][Y] += loc_forces[thread][part][Y];
           }
        }
```

Explain why this change is OK. Run the program with this modification and compare its performance with the original code with block partitioning and the code with a cyclic partitioning of the first phase of the forces calculation. What conclusions can you draw?

6.6. In our discussion of the OpenMP implementation of the basic *n*-body solver, we observed that the implied barrier after the output statement wasn't necessary. We could therefore modify the `single` directive with a `nowait` clause. It's possible to also eliminate the implied barriers at the ends of the two `for`

each particle q loops by modifying for directives with nowait clauses. Would doing this cause any problems? Explain your answer.

6.7. For the shared-memory implementation of the reduced n-body solver, we saw that a cyclic schedule for the computation of the forces outperformed a block schedule, in spite of the reduced cache performance. By experimenting with the OpenMP or the Pthreads implementation, determine the performance of various block-cyclic schedules. Is there an optimal block size for your system?

6.8. If $\mathbf{x}$ and $\mathbf{y}$ are double-precision n-dimensional vectors and α is a double-precision scalar, the assignment

$$\mathbf{y} \longleftarrow \alpha\mathbf{x} + \mathbf{y}$$

is called a DAXPY. DAXPY is an abbreviation of "Double precision Alpha times **X** Plus **Y**." Write a Pthreads or OpenMP program in which the master thread generates two large, random n-dimensional arrays and a random scalar, all of which are doubles. The threads should then carry out a DAXPY on the randomly generated values. For large values of n and various numbers of threads compare the performance of the program with a block partition and a cyclic partition of the arrays. Which partitioning scheme performs better? Why?

6.9. Write an MPI program in which each process generates a large, initialized, m-dimensional array of doubles. Your program should then repeatedly call MPI_Allgather on the m-dimensional arrays. Compare the performance of the calls to MPI_Allgather when the global array (the array that's created by the call to MPI_Allgather) has

a. a block distribution, and

b. a cyclic distribution.

To use a cyclic distribution, download the code cyclic_derived.c from the book's web site, and use the MPI datatype created by this code for the *destination* in the calls to MPI_Allgather. For example, we might call

```
MPI_Allgather(sendbuf, m, MPI_DOUBLE, recvbuf, 1, cyclic_mpi_t,
      comm);
```

if the new MPI datatype were called cyclic_mpi_t.

Which distribution performs better? Why? Don't include the overhead involved in building the derived datatype.

6.10. Consider the following code:

```
int n, thread_count, i, chunksize;
double x[n], y[n], a;
    . . .
#   pragma omp parallel num_threads(thread_count) \
       default(none) private(i) \
       shared(x, y, a, n, thread_count, chunksize)
```

```
    {
#     pragma omp for schedule(static, n/thread_count)
      for (i = 0; i < n; i++) {
         x[i] = f(i);  /* f is a function */
         y[i] = g(i);  /* g is a function */
      }
#     pragma omp for schedule(static, chunksize)
      for (i = 0; i < n; i++)
         y[i] += a*x[i];
   }  /*  omp parallel */
```

Suppose `n = 64`, `thread_count = 2`, the cache-line size is 8 `doubles`, and each core has an L2 cache that can store 131,072 `doubles`. If `chunksize = n/thread_count`, how many L2 cache misses do you expect in the second loop? If `chunksize = 8`, how many L2 misses do you expect in the second loop? You can assume that both `x` and `y` are aligned on a cache-line boundary. That is, both `x[0]` and `y[0]` are the first elements in their respective cache lines.

6.11. Write an MPI program that compares the performance of `MPI_Allgather` using `MPI_IN_PLACE` with the performance of `MPI_Allgather` when each process uses separate send and receive buffers. Which call to `MPI_Allgather` is faster when run with a single process? What if you use multiple processes?

6.12. **a.** Modify the basic MPI implementation of the *n*-body solver so that it uses a separate array for the local positions. How does its performance compare with the performance of the original *n*-body solver? (Look at performance with I/O turned off.)

b. Modify the basic MPI implementation of the *n*-body solver so that it distributes the masses. What changes need to be made to the communications in the program? How does the performance compare with the original solver?

6.13. Using Figure 6.6 as a guide, sketch the communications that would be needed in an "obvious" MPI implementation of the reduced *n*-body solver if there were three processes, six particles, and the solver used a cyclic distribution of the particles.

6.14. Modify the MPI version of the reduced *n*-body solver so that it uses two calls to `MPI_Sendrecv_replace` for each phase of the ring pass. How does the performance of this implementation compare to the implementation that uses a single call to `MPI_Sendrecv_replace`?

6.15. A common problem in MPI programs is converting global array indexes to local array indexes and vice-versa.

a. Find a formula for determining a global index from a local index if the array has a block distribution.

b. Find a formula for determining a local index from a global index if the array has a block distribution.

c. Find a formula for determining a global index from a local index if the array has a cyclic distribution.
d. Find a formula for determining a local index from a global index if the array has cyclic distribution.

You can assume that the number of processes evenly divides the number of elements in the global array. Your solutions should only use basic arithmetic operators ($+, -, *, /$). They shouldn't use any loops or branches.

6.16. In our implementation of the reduced n-body solver, we make use of a function `First_index` which, given a global index of a particle assigned to one process, determines the "next higher" global index of a particle assigned to another process. The input arguments to the function are the following:

a. The global index of the particle assigned to the first process
b. The rank of the first process
c. The rank of the second process
d. The number of processes

The return value is the global index of the second particle. The function assumes that the particles have a cyclic distribution among the processes. Write C-code for `First_index`. (*Hint*: Consider two cases: the rank of the first process is less than the rank of the second, and the rank of the first is greater than or equal to the rank of the second).

6.17. **a.** Use Figure 6.10 to determine the maximum number of records that would be on the stack at any one time in solving a four-city TSP. (*Hint*: Look at the stack after branching as far as possible to the left).
b. Draw the tree structure that would be generated in solving a five-city TSP.
c. Determine the maximum number of records that would be on the stack at any one time during a search of this tree.
d. Use your answers to the preceding parts to determine a formula for the maximum number of records that would be on the stack at any one time in solving an n-city TSP.

6.18. Breadth-first search can be implemented as an iterative algorithm using a **queue**. Recall that a queue is a "first-in first-out" list data structure, in which objects are removed, or *dequeued*, in the same order in which they're added, or *enqueued.* We can use a queue to solve TSP and implement breadth-first search as follows:

```
queue = Init_queue();  /* Create empty queue */
tour = Init_tour();    /* Create partial tour that visits
    hometown */
Enqueue(queue, tour);
while (!Empty(queue)) {
   tour = Dequeue(queue);
   if (City_count(tour) == n) {
      if (Best_tour(tour))
         Update_best_tour(tour);
   } else {
```

```
      for each neighboring city
         if (Feasible(tour, city)) {
            Add_city(tour, city);
            Enqueue(tour);
            Remove_last_city(tour);
         }
   }
   Free_tour(tour);
} /* while !Empty */
```

This algorithm, although correct, will have serious difficulty if it's used on a problem with more than 10 or so cities. Why?

In the shared-memory implementations of our solutions to TSP, we use breadth-first search to create an initial list of tours that can be divided among the threads.

a. Modify this code so that it can be used by thread 0 to generate a queue of at least `thread_count` tours.

b. Once the queue has been generated by thread 0, write pseudocode that shows how the threads can initialize their stacks with a block of tours stored in the queue.

6.19. Modify the Pthreads implementation of static tree search so that it uses a read-write lock to protect the examination of the best tour. Read-lock the best tour when calling `Best_tour`, and write-lock it when calling `Update_best_tour`. Run the modified program with several input sets. How does the change affect the overall run-time?

6.20. Suppose the stack on process/thread A contains k tours.

a. Perhaps the simplest strategy for implementing stack splitting in TSP is to pop $k/2$ tours from A's existing stack and push them onto the new stack. Explain why this is unlikely to be a good strategy.

b. Another simple strategy is to split the stack on the basis of the cost of the partial tours on the stack. The least-cost partial tour goes to A. The second cheapest tour goes to `new_stack`. The third cheapest goes to A, and so on. Is this likely to be a good strategy? Explain your answer.

c. A variation on the strategy outlined in the preceding problem is to use average cost per edge. In average cost per edge, the partial tours on A's stack are ordered according to their cost divided by the number of edges in the partial tour. Then the tours are assigned in round-robin fashion to the stacks, that is, the cheapest cost per edge to A, the next cheapest cost per edge to `new_stack`, and so on. Is this likely to be a good strategy? Explain your answer.

Implement the three strategies outlined here in one of the dynamic load-balancing codes. How do these strategies compare to each other and the strategy outlined in the text? How did you collect your data?

6.21. **a.** Modify the static MPI TSP program so that each process uses a local best tour data structure until it has finished searching. When all the processes

have finished executing, the processes should execute a global reduction to find the least-cost tour. How does the performance of this implementation compare to the static implementation? Can you find input problems for which its performance is competitive with the original static implementation?

b. Create a TSP digraph in which the initial tours assigned to processes $1, 2, \ldots,$ `comm_sz` $- 1$ all have an edge that has a cost that is much greater than the total cost of any complete tour that will be examined by process 0. How do the various implementations perform on this problem when `comm_sz` processes are used?

6.22. `MPI_Recv` and each of the sends we've studied is blocking. `MPI_Recv` won't return until the message is received, and the various sends won't return until the message is sent or buffered. Thus, when one of these operations returns, you know the status of the message buffer argument. For `MPI_Recv`, the message buffer contains the received message—at least if there's been no error—and for the send operations, the message buffer can be reused. MPI also provides a **nonblocking** version of each of these functions, that is, they return as soon as the MPI run-time system has registered the operation. Furthermore, when they return, the message buffer argument cannot be accessed by the user program: the MPI run-time system can use the actual user message buffer to store the message. This has the virtue that the message doesn't have to be copied into or from an MPI-supplied storage location.

When the user program wants to reuse the message buffer, she can force the operation to complete by calling one of several possible MPI functions. Thus, the nonblocking operations split a communication into two phases:

- Begin the communication by calling one of the nonblocking functions
- Complete the communication by calling one of the completion functions

Each of the nonblocking send initiation functions has the same syntax as the blocking function, except that there is a final *request* argument. For example,

```
int MPI_Isend(
      void*          msg         /* in  */,
      int            count       /* in  */,
      MPI_Datatype   datatype    /* in  */,
      int            dest        /* in  */,
      int            tag         /* in  */,
      MPI_Comm       comm        /* in  */,
      MPI_Request*   request_p   /* out */);
```

The nonblocking receive replaces the status argument with a request argument. The request arguments identify the operation to the run-time system, so that when a program wishes to complete the operation, the completion function takes a request argument.

The simplest completion function is MPI_Wait:

```
int MPI_Wait(
      MPI_Request*  request_p  /* in/out */
      MPI_Status*   status_p   /* out    */);
```

When this returns, the operation that created *request_p will have completed. In our setting, *request_p will be set to MPI_REQUEST_NULL, and *status_p will store information on the completed operation.

Note that nonblocking receives can be matched to blocking sends and nonblocking sends can be matched to blocking receives.

We can use nonblocking sends to implement our broadcast of the best tour. The basic idea is that we create a couple of arrays containing comm_sz elements. The first stores the cost of the new best tour, the second stores the requests, so the basic broadcast looks something like this:

```
int costs[comm_sz];
MPI_Request requests[comm_sz];

for (dest = 0; dest < comm_sz; dest++)
   if (my_rank != dest) {
      costs[dest] = new_best_tour_cost;
      MPI_Isend(&costs[dest], 1, MPI_INT, dest, NEW_COST_TAG,
            comm, &requests[dest]);
   }
requests[my_rank] = MPI_REQUEST_NULL;
```

When this loop is completed, the sends will have been started, and they can be matched by ordinary calls to MPI_Recv.

There are a variety of ways to deal with subsequent broadcasts. Perhaps the simplest is to wait on all the previous nonblocking sends with the function MPI_Waitall:

```
int MPI_Waitall(
      int          count       /* in     */,
      MPI_Request  requests[]  /* in/out */,
      MPI_Status   statuses[]  /* out    */);
```

When this returns, all of the operations will have completed (assuming there are no errors). Note that it's OK to call MPI_Wait and MPI_Waitall if a request has the value MPI_REQUEST_NULL.

Use nonblocking sends to implement a broadcast of best tour costs in the static MPI implementation of the TSP program. How does its performance compare to the performance of the implementation that uses buffered sends?

6.23. Recall that an MPI_Status object is a struct with members for the source, the tag, and any error code for the associated message. It also stores information

on the size of the message. However, this isn't directly accessible as a member, it is only accessible through the MPI function `MPI_Get_count`:

```
int MPI_Get_count(
      MPI_Status*   status_p   /* in  */,
      MPI_Datatype  datatype   /* in  */,
      int*          count_p    /* out */);
```

When `MPI_Get_count` is passed the status of a message and a datatype, it returns the number of objects of the given datatype in the message. Thus, `MPI_Iprobe` and `MPI_Get_count` can be used to determine the size of an incoming message before the message is received. Use these to write a `Cleanup_messages` function that can be called before an MPI program quits. The purpose of the function is to receive any unreceived messages so that functions such as `MPI_Buffer_detach` won't hang.

6.24. The program `mpi_tsp_dyn.c` takes a command-line argument `split_cutoff`. If a partial tour has visited `split_cutoff` or more cities, it's not considered a candidate for sending to another process. The `Fulfill_request` function will therefore only send partial tours with fewer than `split_cutoff` cities. How does `split_cutoff` affect the overall run-time of the program? Can you find a reasonably good rule of thumb for deciding on the `split_cutoff`? How does changing the number of processes (that is, changing `comm_sz`) affect the best value for `split_cutoff`?

6.25. Pointers cannot be sent by MPI programs since an address that is valid on the sending process may cause a segmentation violation on the receiving process, or, perhaps worse, refer to memory that's already being used by the receiving process. There are a couple of alternatives that can be used to address this problem:

a. The object that uses pointers can be packed into contiguous memory by the sender and unpacked by the receiver.

b. The sender and the receiver can build MPI derived datatypes that map the memory used by the sender and valid memory on the receiver.

Write two `Send_linked_list` functions and two matching `Recv_linked_list` functions. The first pair of send-receive functions should use `MPI_Pack` and `MPI_Unpack`. The second pair should use derived datatypes. Note that the second pair may need to send two messages: the first will tell the receiver how many nodes are in the linked list, and the second will send the actual list. How does the performance of the two pairs of functions compare? How does their performance compare to the cost of sending a block of contiguous memory of the same size as the packed list?

6.26. The dynamically partitioned MPI implementation of the TSP solver uses a termination detection algorithm that may require the use of very high-precision

rational arithmetic (that is, common fractions with very large numerators and/or denominators).

a. If the total amount of energy is `comm_sz`, explain why the amount of energy stored by any process other than zero will have the form $1/2^k$ for some nonnegative integer k. Thus, the amount of energy stored by any process other than zero can be represented by k, an unsigned integer.

b. Explain why the representation in the first part is extremely unlikely to overflow or underflow.

c. Process 0, on the other hand, will need to store fractions with a numerator other than one. Explain how to implement such a fraction using an unsigned integer for the denominator and a bit array for the numerator. How can this implementation deal with overflow of the numerator?

6.27. If there are many processes and many redistributions of work in the dynamic MPI implementation of the TSP solver, process 0 could become a bottleneck for energy returns. Explain how one could use a spanning tree of processes in which a child sends energy to its parent rather than process 0.

6.28. Modify the implementation of the TSP solver that uses MPI and dynamic partitioning of the search tree so that each process reports the number of times it sends an "out of work" message to process 0. Speculate about how receiving and handling the out of work messages affects the overall run-time for process 0.

6.29. The C source file `mpi_tsp_dyn.c` contains the implementation of the MPI TSP solver that uses dynamic partitioning of the search tree. The online version uses the first of the three methods outlined in Section 6.2.12 for determining to which process a request for work should be sent. Implement the other two methods and compare the performance of the three. Does one method consistently outperform the other two?

6.30. Determine which of the three APIs is preferable for the n-body solvers and solving TSP.

a. How much memory is required for each of the serial programs? When the parallel programs solve large problems, will they fit into the memory of your shared-memory system? What about your distributed-memory system?

b. How much communication is required by each of the parallel algorithms?

c. Can the serial programs be easily parallelized by the use of OpenMP directives? Do they need synchronization constructs such as condition variables or read-write locks?

Compare your decisions with the actual performance of the programs. Did you make the right decisions?

6.7 PROGRAMMING ASSIGNMENTS

6.1. Look up the classical fourth-order Runge Kutta method for solving an ordinary differential equation. Use this method instead of Euler's method to estimate

the values of $\mathbf{s}_q(t)$ and $\mathbf{s}'_q(t)$. Modify the reduced versions of the serial n-body solver, either the Pthreads or the OpenMP n-body solver, and the MPI n-body solver. How does the output compare to the output using Euler's method? How does the performance of the two methods compare?

6.2. Modify the basic MPI n-body solver so that it uses a ring pass the instead of a call to `MPI_Allgather`. When a process receives the positions of particles assigned to another process, it computes *all* the forces resulting from interactions between its assigned particles and the received particles. After receiving `comm_sz` $-$ 1 sets of positions, each process should be able to compute the total force on each of its particles. How does the performance of this solver compare with the original basic MPI solver? How does its performance compare with the reduced MPI solver?

6.3. We can simulate a ring pass using shared-memory:

```
Compute loc_forces and tmp_forces due to my particle
    interactions;
Notify dest that tmp_forces are available;
for (phase = 1; phase < thread_count; phase++) {
   Wait for source to notify me that tmp_forces are available;
   Compute forces due to my particle interactions with
         ''received'' particles;
   Notify dest that tmp_forces are available;
}
Add my tmp_forces into my loc_forces;
```

To implement this, the main thread can allocate n storage locations for the total forces and n locations for the "temp" forces. Each thread will operate on the appropriate subset of locations in the two arrays. It's easiest to implement "notify" and "wait" using semaphores. The main thread can allocate a semaphore for each source-dest pair and initialize each semaphore to 0 (or "locked"). After a thread has computed the forces, it can call `sem_post` to notify the dest thread, and a thread can block in a call to `sem_wait` to wait for the availability of the next set of forces. Implement this scheme in Pthreads. How does its performance compare with the performance of the original reduced OpenMP/Pthreads solver? How does its performance compare with the reduced MPI solver? How does its memory usage compare with the reduced OpenMP/Pthreads solver? The reduced MPI solver?

6.4. The storage used in the reduced MPI n-body solver can be further reduced by having each process store only its n/`comm_sz` masses and communicating masses as well as positions and forces. This can be implemented by adding storage for an additional n/`comm_sz` `doubles` to the `tmp_data` array. How does this change affect the performance of the solver? How does the memory required by this program compare with the memory required for the original MPI n-body solver? How does its memory usage compare with the reduced OpenMP solver?

6.5. The `Terminated` function in the OpenMP dynamic implementation of tree search uses busy-waiting, which can be very wasteful of system resources. Ask a system guru if your Pthreads and OpenMP implementations can be used together in a single program. If so, modify the solution in the OpenMP dynamic implementation so that it uses Pthreads and condition variables for work redistribution and termination. How does the performance of this implementation compare with the performance of the original implementation?

6.6. The implementations of iterative tree search that we discussed used an array-based stack. Modify the implementation of either the Pthreads or OpenMP dynamic tree search program so that it uses a linked-list-based stack. How does the use of the linked list affect the performance?

Add a command-line argument for the "cutoff size" that was discussed briefly in the text. How does the use of a cutoff size affect the performance?

6.7. Use Pthreads or OpenMP to implement tree search in which there's a shared stack. As we discussed in the text, it would be very inefficient to have all calls to `Push` and `Pop` access a shared stack, so the program should also use a local stack for each thread. However, the `Push` function can occasionally push partial tours onto the shared stack, and the `Pop` function can pop several tours from the shared stack and push them onto the local stack, if the calling thread has run out of work. Thus, the program will need some additional input arguments:

- **a.** The frequency with which tours are pushed onto the shared stack. This can be an `int`. For example, if every 10th tour generated by a thread should be pushed onto the shared stack, then the command-line argument would be 10.
- **b.** A blocksize for pushing. There may be less contention if, rather than pushing a single tour onto the shared stack, a block of several tours is pushed.
- **c.** A blocksize for popping. If we pop a single tour from the shared stack when we run out of work, we may get too much contention for the shared stack.

How can a thread determine whether the program has terminated?

Implement this design and your termination detection with Pthreads or OpenMP. How do the various input arguments affect its performance? How does the optimal performance of this program compare with the optimal performance of the dynamically load-balanced code we discussed in the text?

CHAPTER

7 Where to Go from Here

Now that you know the basics of writing parallel programs using MPI, Pthreads, and OpenMP, you may be wondering if there are any new worlds left to conquer. The answer is a resounding yes. Here are a few topics for further study. With each topic we've listed several references. Keep in mind, though, that this list is necessarily brief, and parallel computing is a fast-changing field. You may want to do a little searching on the Internet before settling down to do more study.

1. *MPI*. MPI is a large and evolving standard. We've only discussed a part of the original standard, MPI-1. We've learned about some of the point-to-point and collective communications and some of the facilities it provides for building derived datatypes. MPI-1 also provides standards for creating and managing communicators and topologies. We briefly discussed communicators in the text; recall that roughly speaking, a communicator is a collection of processes that can send messages to each other. Topologies provide a means for imposing a logical organization on the processes in a communicator. For example, we talked about partitioning a matrix among a collection of MPI processes by assigning blocks of rows to each process. In many applications, it's more convenient to assign block submatrices to the processes. In such an application, it would be very useful to think of our processes as a rectangular grid in which a process is identified with the submatrix it's working with. Topologies provide a means for us to make this identification. Thus, we might refer to the process in, say, the second row and fourth column instead of process 13. The texts [23, 43, 47] all provide more in-depth introductions to MPI-1.

   MPI-2 added dynamic process management, one-sided communications, and parallel I/O to MPI-1. We mentioned one-sided communications and parallel I/O in Chapter 2. Also, when we discussed Pthreads, we mentioned that many Pthreads programs create threads as they're needed rather than creating all the threads at the beginning of execution. Dynamic process management adds this and other capabilities to MPI process management. [24] is an introduction to MPI-2. There is also a combined edition of the MPI-1 and MPI-2 standards [37].
2. *Pthreads and semaphores*. We've discussed some of the functions Pthreads provides for starting and terminating threads, protecting critical sections, and

An Introduction to Parallel Programming. DOI: 10.1016/B978-0-12-374260-5.00007-5

synchronizing threads. There are a number of other functions, and it's also possible to change the behavior of most of the functions we've looked at. Recall that the various object initialization functions take an "attribute" argument. We always simply passed in `NULL` for these arguments, so the object functions would have the "default" behavior. Passing in other values for these arguments changes this. For example, if a thread obtains a mutex and it then attempts to relock the mutex (e.g., when it makes a recursive function call) the default behavior is undefined. However, if the mutex was created with the attribute `PTHREAD_MUTEX_ERRORCHECK`, the second call to `pthread_mutex_lock` will return an error. On the other hand, if the mutex was created with the attribute `PTHREAD_MUTEX_RECURSIVE`, then the attempt to relock will succeed.

Another subject we've only touched on is the use of nonblocking operations. We mentioned `pthread_mutex_trylock` when we were discussing the dynamic implementation of tree search. There are also nonblocking versions of the read-write lock functions and `sem_wait`. These functions provide the opportunity for a thread to continue working when a lock or semaphore is owned by another thread. They therefore have the potential to greatly increase the parallelism in an application.

Like MPI, Pthreads and semaphores are evolving standards. They are part of a collection of standards known as POSIX. The latest version of POSIX is available online through The Open Group [41]. The Pthreads header file man page [46] and the semaphores header file man page [48] provide links to man pages for all the Pthreads and semaphore functions. For some texts on Pthreads, see [6, 32].

3. *OpenMP*. We've learned about some of the most important directives, clauses, and functions in OpenMP. We've learned about how to start multiple threads, how to parallelize `for` loops, how to protect critical sections, how to schedule loops, and how to modify the scope of variables. However, there is still a good deal more to learn. Perhaps the most important new directive is the recently introduced `task` directive. It can be used to parallelize such constructs as recursive function calls and `while` loops. In essence, it identifies a structured block as a task that can be executed by a thread. When a thread encounters a `task` directive, the thread can either execute the code in the structured block or it can be added to a conceptual pool of tasks, and threads in the current team will execute the tasks until the pool is empty.

   The OpenMP Architecture Review Board is continuing development of the OpenMP standard. The latest documents are available at [42]. For some texts, see [8, 10, 47].
4. *Parallel hardware*. Parallel hardware tends to be a fast-moving target. Fortunately, the texts by Hennessy and Patterson [26, 44] are updated fairly frequently. They provide a comprehensive overview on topics such as instruction-level parallelism, shared-memory systems, and interconnects. For a book that focuses exclusively on parallel systems, see [11].
5. *General parallel programming*. There are a number of books on parallel programming that don't focus on a particular API. [50] provides a relatively elementary

discussion of both distributed- and shared-memory programming. [22] has an extensive discussion of parallel algorithms and a priori analysis of program performance. [33] provides an overview of current parallel programming languages, and some directions that may be taken in the near future.

For discussions of shared-memory programming see [3, 4, 27]. The text [4] discusses techniques for designing and developing shared-memory programs. In addition it develops a number of parallel programs for solving problems such as searching and sorting. Both [3] and [27] provide in-depth discussions of determining whether an algorithm is correct and mechanisms for insuring correctness.

6. *GPUs*. One of the most promising developments in parallel computing is the use of graphics processing units (GPUs) for general purpose parallel computing. They have been used very successfully in a variety of data parallel programs, or programs that obtain parallelism by partitioning the data among the processors. The article [17] provides an overview of GPUs. The book [30] also provides an overview of GPUs. In addition, it gives an introduction to programming them using CUDA, NVIDIA's language for programming GPUs, and OpenCL, a standard for programming heterogeneous systems including conventional CPUs and GPUs.
7. *History of parallel computing*. It's surprising to many programmers that parallel computing has a long and venerable history—well, as long and venerable as most topics in computer science. The article [14] provides a very brief survey and some references. The website [51] is a timeline listing milestones in the development of parallel systems.

Now go out and conquer new worlds.

This page intentionally left blank

References

[1] S.V. Adve, K. Gharachorloo, Shared memory consistency models: A tutorial Digital Western Research Laboratory research report 95/7. <ftp://gate-keeper.dec.com/pub/DEC/WRL/research-reports/WRL-TR-95.7.pdf>, 1995 (accessed 21.09.10).

[2] G.M. Amdahl, Validity of the single processor approach to achieving large scale computing capabilities, in: Proceedings of the American Federation of Information Processing Societies Conference, vol. 30, issue 2, Atlantic City, NJ, 1967, pp. 483–485.

[3] G.R. Andrews, Multithreaded, Parallel, and Distributed Programming, Addison-Wesley, Boston, 2000.

[4] C. Breshears, The Art of Concurrency: A Thread Monkeys Guide to Writing Parallel Applications, O'Reilly, Sebastopol, CA, 2009.

[5] R.E. Bryant, D.R. O'Hallaron, Computer Systems: A Programmer's Perspective, Prentice Hall, Englewood Cliffs, NJ, 2002.

[6] D.R. Butenhof, Programming with Posix Threads, Addison-Wesley, Reading, MA, 1997.

[7] B.L. Chamberlain, D. Callahan, H.P. Zima, Parallel programmability and the Chapel language, Int. J. High Perform. Comput. Appl. 21 (3) (2007) 291–312. SAGE Publications, Thousand Oaks, CA, see also <http://chapel.cray.com/> (accessed 21.09.10).

[8] R. Chandra, et al., Parallel Programming in OpenMP, Morgan Kaufmann, San Francisco, 2001.

[9] P. Charles, et al., X10: An object-oriented approach to non-uniform clustered computing, in: R.E. Johnson, R.P. Gabriel (Eds.), Proceedings of the 20th Annual ACM Special Interest Group on Programming Languages (SIGPLAN) Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), San Diego, 16–20 October, ACM, New York, 2005, pp. 519–538. See also <http://x10-lang.org/> (accessed 21.09.10).

[10] B. Chapman, G. Jost, R. van der Pas, Using OpenMP: Portable Shared Memory Parallel Programming, MIT Press, Cambridge, MA, 2008.

[11] D. Culler, J.P. Singh, A. Gupta, Parallel Computer Architecture: A Hardware/Software Approach, Morgan Kaufmann, San Francisco, 1998.

[12] J. Demmel, Hierarchical methods for the *N*-body problem, Lecture 22 for Demmel's spring 2009 class, "Applications of Parallel Computing," CS267/EngC233, UC Berkeley. <http://www.cs.berkeley.edu/~demmel/cs267_Spr09/Lectures/lecture22_NBody_jwd09.ppt>, 2009 (accessed 21.09.10).

[13] E.W. Dijkstra, Cooperating sequential processes, in: F. Genuys (Ed.), Programming Languages, Academic Press, New York, 1968, pp. 43–112. A draft version of this paper is available from <http://userweb.cs.utexas.edu/users/EWD/ewd01xx/EWD123.PDF> (accessed 21.09.10).

[14] P.J. Denning, J.B. Dennis, The resurgence of parallelism, Commun. ACM 53 (6) (2010) 30–32.

[15] E.N. Dorband, M. Hemsendorf, D. Merritt, Systolic and hyper-systolic algorithms for the gravitational N-body problem, with an application to Brownian motion, J. Comput. Phys. 185 (2003) 484–511.

[16] Eclipse Parallel Tools Platform. <http://www.eclipse.org/ptp/> (accessed 21.09.10).

[17] K. Fatahalian, M. Houston, A closer look at GPUs, Commun. ACM 51 (10) (2008) 50–57.

[18] M. Flynn, Very high-speed computing systems, Proc. IEEE 54 (1966) 1901–1909.

[19] I. Foster, Designing and Building Parallel Programs, Addison-Wesley, Reading, MA, 1995. Also available from <http://www.mcs.anl.gov/~itf/dbpp/> (accessed 21.09.10).
[20] I. Foster, C. Kesselman, (Eds.), The Grid 2, second ed., Blueprint for a New Computing Infrastructure, Morgan Kaufmann, San Francisco, 2003.
[21] The GNU MP Bignum Library. <http://gmplib.org/> (accessed 21.09.10)
[22] A. Grama, et al., Introduction to Parallel Computing, second ed., Addison-Wesley, Harlow, Essex, 2003.
[23] W. Gropp, E. Lusk, A. Skjellum, Using MPI, second ed., MIT Press, Cambridge, MA, 1999.
[24] W. Gropp, E. Lusk, R. Thakur, Using MPI-2, MIT Press, Cambridge, MA, 1999.
[25] J.L. Gustafson, Reevaluating Amdahl's law, Commun. ACM 31 (5) (1988) 532–533.
[26] J. Hennessy, D. Patterson, Computer Architecture: A Quantitative Approach, fourth ed., Morgan Kaufmann, San Francisco, 2006.
[27] M. Herlihy, N. Shavit, The Art of Multiprocessor Programming, Morgan Kaufmann, Boston, 2008.
[28] IBM, IBM InfoSphere streams v1.2.0 supports highly complex heterogeneous data analysis, IBM United States Software Announcement 210-037, February 23, 2010. Available from <http://www.ibm.com/common/ssi/rep_ca/7/897/ENUS210-037/ENUS210-037.PDF> (accessed 21.09.10).
[29] B.W. Kernighan, D.M. Ritchie, The C Programming Language, second ed., Prentice-Hall, Upper Saddle River, NJ, 1988.
[30] D.B. Kirk, W-m.W. Hwu, Programming Massively Parallel Processors: A Hands-on Approach, Morgan Kaufmann, Boston, 2010.
[31] J. Larus, C. Kozyrakis, Transactional memory, Commun. ACM 51 (7) (2008) 80–88.
[32] B. Lewis, D.J. Berg, Multithreaded Programming with Pthreads, Prentice-Hall, Upper Saddle River, NJ, 1998.
[33] C. Lin, L. Snyder, Principles of Parallel Programming, Addison-Wesley, Boston, 2009.
[34] J. Makino, An efficient parallel algorithm for $O(N^2)$ direct summation method and its variations on distributed-memory parallel machines. New Astron. 7 (2002) 373–384.
[35] J.M. May, Parallel I/O for High Performance Computing, Morgan Kaufmann, San Francisco, 2000.
[36] Message-Passing Interface Forum. <http://www.mpi-forum.org> (accessed 21.09.10).
[37] Message-Passing Interface Forum, MPI: A message-passing interface standard, version 2.2. Available from <http://www.mpi-forum.org/docs/mpi-2.2/mpi22-report.pdf> (accessed 21.09.10).
[38] Microsoft Visual Studio 2010. <http://msdn.microsoft.com/en-us/vstudio/default.aspx> (accessed 21.09.10).
[39] Message-Passing Interface Forum, MPI: A message-passing interface standard. Available from <http://www.mpi-forum.org/docs/mpi-11-html/mpi-report.html> (accessed 21.09.10).
[40] L. Oliker, et al., Effects of ordering strategies and programming paradigms on sparse matrix computations, SIAM Rev. 44 (3) (2002) 373–393.
[41] The Open Group. <http://www.opengroup.org> (accessed 21.09.10).
[42] OpenMP Architecture Review Board, OpenMP application program interface, version 3.0, May 2008. See also <http://openmp.org/wp/> (accessed 21.09.10).
[43] P. Pacheco, Parallel Programming with MPI, Morgan Kaufmann, San Francisco, 1997.

[44] D.A. Patterson, J.L. Hennessy, Computer Organization and Design: The Hardware/Software Interface, fourth ed., Morgan Kaufmann, Boston, 2009.
[45] Project Fortress Community. Available from <http://projectfortress.sun.com/Projects/Community> (accessed 21.09.10).
[46] `pthread.h` manual page. Available from <http://www.opengroup.org/onlinepubs/9699919799/basedefs/pthread.h.html> (accessed 21.09.10).
[47] M. Quinn, Parallel Programming in C with MPI and OpenMP, McGraw-Hill Higher Education, Boston, 2004.
[48] `semaphore.h` manual page. Available from <http://www.opengroup.org/onlinepubs/000095399/basedefs/semaphore.h.html> (accessed 21.09.10).
[49] Valgrind home. <http://valgrind.org> (accessed 21.09.10).
[50] B. Wilkinson, M. Allen, Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers, second ed., Prentice-Hall, Upper Saddle River, NJ, 2004.
[51] G. Wilson, The history of the development of parallel computing. Available from <http://parallel.ru/history/wilson_history.html>, 1994 (accessed 21.09.10).

This page intentionally left blank

Index

A

Agglomerate (as in Foster's Methodology), 278
Aggregate (as in Foster's Methodology), 76
Amdahl's law, 61–62
Application programming interface (API), 32, 53–54, 71, 153, 335–336
Argument aliasing, 106, 137
Arithmetic and logic unit (ALU), 16, 30
Asynchronous, 32
`atomic` directives, 245–246, 249

B

Back-substitution, 269
Bandwidth, 38, 42, 74
Barriers, 176–179, 199
  implementing, 177, 178
Best tour
  maintaining, 321–325
  printing, 325–326
`Best_tour` function, 310
Bisection bandwidth, 38, 74
Bisection width, 37, 38, 74
Block partition of vector, 109, 138
Block-cyclic partition of vectors, 110, 138
Breadth-first search, 307
Broadcast, 106–109
  tree-structured, 108*f*
Bubble sort, 232–233
Buffered send, 323–325, 340
Bus, 16, 35
Busy-waiting, 165–168, 172*t*, 177, 199
Butterfly, 106, 107*f*

C

C compiler, 85
Cache, 19, 21*t*
  blocks, 19, 191, 200
  coherence, 43–44, 251–256
    directory-based, 44–45
    false sharing, 45–46
    problem, 191
    snooping, 44
    use of, 191
  CPU, 19, 22
  direct mapped, 21
  fully associative, 20
  hit, 20
  lines, 19, 191, 200
  mappings, 20–22
  memory, 191, 251
  miss, 20
  profiler, 193
  and programs, 22–23
  read, 20
  write, 20
  write-back, 20
  write-through, 20
Caching, basics of, 19–20
Central processing unit (CPU), 15–17
  cache, 19, 22
`Check_for_work` function, 334
Chunk, 238, 239
`Chunksize`, 238, 239, 261
Clause, 214, 259
Climate modeling, 2
`Clock` function (C library), 121, 148
Clusters, 35
Coarse-grained multithreading, 29
Coarser-grained parallelism, 28
Collective communication, 101, 137
  definition, 103
  point-to-point *vs.*, 105–106
Column-oriented algorithm, 269–270
Command line, 70, 212
Command line option, 211n
Commit (MPI datatype), 119
Communication, 7, 12, 88
  collective, 101, 137
  `MPI_Reduce`, 103–105, 105*t*
  ping-pong, 148
  point-to-point, 104, 137
  for trapezoidal rule, 96*f*
  tree-structured, 102–103

Page numbers followed by "*f*" indicates a figure, "*t*" indicates a table, and "n" indicates footnote.

Communicator, MPI, 87–88, 136
Compiler, 28, 71, 153, 166, 227
Compiler use, 211n
Computational power, 2
Concurrent computing, 9–10
Condition variables, 179–181, 199
  condition broadcast, 199
  condition signal, 199
  condition wait, 190
Consumer threads, 242
Coordinating processes/threads, 48–49
Core, 3
Count sort, 273
CPU, *see* Central processing unit
`critical` directives, 249
Critical sections, 162–165, 168, 199, 218, 223
  and busy-waiting, 165–168
  and locks, 246–248
  and mutexes, 168–171
  and semaphores, 174
Crossbar, 35, 40, 74
CUDA, 355
Cyclic partition of vector, 109, 138

D

Data
  analysis, 2
  dependences, 227–228
  distributions, 109–110
  parallelism, 6–7, 30
DAXPY, 343
Deadlock, 132, 140, 203
Depth-first search, 301–305
Dequeue, 243, 248
Derived datatypes, 116–119
  `Get_input` function with, 120
Dijkstra, Edsger, 174
Direct interconnect, 37–38
Directed graph (digraph), 300
Directives-based shared-memory programming, 210
Directory-based cache coherence, 44–45
Distributed computing, 9–10, 12
Distributed-memory
  interconnects, 37–42
  programs, 53–56
    message-passing APIs, 53–55
    one-sided communication, 55
    partitioned global address space languages, 55–56
  systems, 8, 9*f*, 12, 33*f*, 35, 83*f*
  vs. shared-memory, 46
Distributed termination detection, 331–333
Drug discovery, 2
Dynamic mapping scheme, 308
Dynamic parallelization of tree search using pthreads, 310–315
`dynamic` schedule types, 239
Dynamic threads, 49

E

Eclipse (integrated development environment), 70
Efficiency, 58–61, 75, 125–126, 139, 253
Embarrassingly parallel, 48, 74
Energy (in distributed termination detection), 332
Energy research, 2
Enqueue, 242, 243
Envelope (of message), 93
Error checking, 215–216
Euler, Leonhard, 275
Euler's Method, 275, 276*f*, 279, 350–351
Explicit parallel constructs, 8

F

Fairness, 250
False sharing, 194, 200, 255–256
Feasible (tree search), 302, 307
Fetch, 20, 26
`fgets` function, 196
File-wide scope, 220
`Find_bin` function, 67, 68
Fine-grained multithreading, 29
Flynn's taxonomy, 29
`for` directive (OpenMP), 235, 260
Fork, 158, 259
Forking process, 213
Foster's methodology, 66, 68*f*, 129, 271, 277, 279
Foster's methodology, 216
`Fulfill_request (in parallel tree search)`, 327, 329, 349
Fully-connected network, 38, 39*f*
Function-wide scope, 220

G

Gaussian elimination, 269
`gcc` (GNU C compiler), 85n, 153n
General parallel programming, 354
`Get_rank` function, 53

Get_current_time function, 64
GET_TIME macro, 121, 138, 203
gettimeofday function, 121
Global sum, 5–7
  function, 103
  multiple cores forming, 5
Global variables, 97, 137
Graphics processing units (GPUs), 32, 355
guided schedule types, 239
Gustafson's law, 62

H

Hang, 94, 132
Hardware multithreading, 28–29
Heterogeneous system, 73
Homogeneous system, 73
Hybrid systems, 35
Hypercubes, 39, 40*f*

I

ILP, *see* Instruction-level parallelism
Indirect interconnects, 40, 40*f*
Input and output (I/O), 56–57, 75, 280–281
Instruction-level parallelism (ILP)
  multiple issue, 27–28
  pipelining, 25–26, 27*t*
Integrated development environments (IDEs), 70
Interconnection networks
  direct interconnect, 37–38
  distributed-memory interconnects, 37–42
  indirect interconnects, 40
  latency and bandwidth, 42
  shared-memory interconnects, 35–37
  switched interconnects, 35

J

Joining process, 213

K

Kernighan, Brian, and Ritchie, Dennis, 71, 84
Kluge, 157

L

Latency, 42, 74
Leaf (search tree), 301, 306
Libraries (MPI, Pthreads, OpenMP), 8
Light-weight processes, 152
Linear speedup, 125
Linked list
  functions, 181–183
    Delete, 182, 184
    Insert, 182, 183
    Member, 182, 186
  multithreaded, 183–187
Linux, 153
Load (as in load/store), 31
Load balancing, 7, 12, 48
Local variables, 97, 137
Lock data structure, 242
Locks, 246–248
Loop-carried dependences, 228–229
Loops
  bubble sort, 232–233
  odd-even transposition sort, 233–236
  scheduling, 236–241
lpthread, 153

M

MacOS X, 153
Main memory, 15, 71, 72, 251
Main thread, 157, 158*f*
Man page, 354
Mapping (Foster's methodology), 76, 279
Mapping (Caches), 20–22
Master thread (OpenMP), 214
Matrix-vector multiplication, 114*f*, 159–162, 160*f*, 192
  local, 124
  parallel, 116, 125*t*, 126*t*
  performance of, 119
  results of timing, 122, 123*t*
  run-times and efficiencies of, 192*t*
  serial, 115
Member function, 182
  implementation of, 186
memcpy (C library function), 268
Memory, 15, 16
  cache, 191, 251
  transactional, 52
  virtual, 23–25
memset (C library function), 277
Merge, 135, 147
Mergesort, 148
Message, matching, 91–92
Message-passing, 242
  locks in, 248–249

Message-Passing Interface (MPI), 8, 83, 338–341, 353
communicator, 87–88
datatypes, 89*t*
derived datatypes, 116–119
and dynamic partitioning
checking and receiving, 333–334
distributed termination detection, 331–333, 332*t*
OpenMP, 327
performance of, 334–335, 334*t*
Pthreads, 327
sending requests, 333
splitting stack, 329–331
forum, 106
implementation, 290, 293*f*, 296, 323, 334
input, 100–101
operators in, 104*t*
output, 97–100
parallelizing, *n*-body solvers, 290–297
programs, 86
performance evaluation of, 119–127
potential pitfall in, 94
safety in, 132–135, 140
scatter, 110–112
solvers, performance of, 297–299, 298*t*
and static partitioning
maintaining, 321–325
partitioning, 320–321
printing, 325–326
unreceived messages, 326
trapezoidal rule in, 94–97
Microprocessor, increase in performance, 1–3
Microsoft, 70
MIMD systems, *see* Multiple instruction, multiple data systems
Modes and buffered sends, 323–325
Monitor, 52
Monte Carlo method, 148, 207, 268
MPI, *see* Message-Passing Interface
mpi.h, 86, 136
MPI_Aint, 118
MPI_Allgather, 115, 124
MPI_Allreduce, 106
MPI_ANY_SOURCE, 91, 92, 322, 340
MPI_BAND, 104
MPI_Barrier, 122, 138
MPI_Bcast, 108, 109, 117, 137
MPI_BOR, 104
MPI_Bsend, 324, 340
MPI_BSEND_OVERHEAD, 325
MPI_Buffer_attach, 324, 340
MPI_Buffer_detach, 324, 326, 340
MPI_BXOR, 104
mpicc command, 85
MPI_CHAR, 53, 89
MPI_Comm, 88
MPI_Comm_rank, 87–88
MPI_Comm_size, 87–88
MPI_COMM_WORLD, 87–88
MPI_Datatype, 89, 104
MPI_DOUBLE, 89
MPI_ERROR, 92
mpiexec command, 86
MPI_Finalize, 86, 136
MPI_FLOAT, 89
MPI_Gather, 112, 137
MPI_Gatherv, 143, 321, 339
MPI_Get_address, 118
MPI_Get_count, 92, 93, 349
MPI_Init, 86, 136
MPI_IN_PLACE, 338, 344
MPI_INT, 89, 321
MPI_Iprobe, 322, 326, 329, 331, 339, 340
MPI_Isend, 347
MPI_LAND, 104
MPI_LOR, 104
MPI_LXOR, 104
MPI_MAX, 104, 122
MPI_MAXLOC, 104
MPI_MIN, 326
MPI_MINLOC, 326, 339
MPI_Op, 104
MPI_Pack, 117, 138, 145, 324, 329, 330, 341
MPI_Pack_size, 324
MPI_PACKED, 145, 331, 341
MPI_Recv, 90–91, 136
semantics of, 93–94
MPI_Reduce, 103–105, 105*t*
MPI_Reduce_scatter, 148
MPI_Request, 348
MPI_REQUEST_NULL, 348
MPI_Rsend, 324, 340
MPI_Scan, 142
MPI_Scatter, 111, 137
MPI_Scatterv, 143, 320, 339
MPI_Send, 88–90, 136–137
semantics of, 93–94

MPI_Sendrecv, 133-135, 140, 296
MPI_Sendrecv_replace, 140, 296, 344
MPI_SOURCE, 92, 322, 340
MPI_Send, 132, 140, 324, 340
MPI_Status, 92, 93, 348
MPI_STATUS_IGNORE, 91
MPI_SUM, 104, 105
MPI_TAG, 92
MPI_Type_commit, 119
MPI_Type_contiguous, 143
MPI_Type_create_struct, 118, 138, 144
MPI_Type_free, 119
MPI_Type_indexed, 144
MPI_Type_vector, 143, 144
MPI_Unpack, 145, 329, 330, 340, 349
MPI_Wait, 348
MPI_Waitall, 348
MPI_Wtime function, 64
Multithreaded linked list, 183–187
Multithreaded programming, 153
Multicores
  forming global sum, 5*f*
  integrated circuits, 12
  processors, 3
Multiple instruction, multiple data (MIMD) systems, 32–33
  distributed-memory system, 33, 33*f*, 35
  shared-memory systems, 33–34, 33*f*
Multiprocessor, 1
Multistage interconnect, 74
Multitasking operating system, 17–18
Mutexes, 51, 168–171, 177, 199
Mutual exclusion techniques, caveats, 249–251
My_avail_tour_count, 329

N

n-body solvers
  I/O, 280–281
  MPI solvers, performance of, 297–299, 298*t*
  OpenMP Codes evaluation, 288–289, 288*t*
  parallelizing
    communications, 278*f*, 279*f*
    computation, 280
    Foster's methodology, 277, 279
    MPI, 290–297
    OpenMP, 281, 284, 288–289
    Pthreads, 289–290
  problem, 271–272
  two serial programs
    basic and reduced algorithm, 274
    compute forces, 273, 274
    Euler's Method, 275, 276*f*
    numerical method, 274
    tangent line, 275, 275*f*
Nested for loops, 124
Nonblocking, 337, 347
Nondeterminism, 49–52, 74, 100
Nonovertaking, 93
Nonrecursive depth-first search, 303–305
Nonuniform memory access (NUMA) system, 34, 34*f*
nowait clause, 284, 338
NP complete, 299
num_threads clause, 214
NVIDIA, 355

O

Odd-even transposition sort, 233–236
  algorithm, 128
    parallel, 129–132, 131*t*, 134–136, 135*t*
omp.h, 212, 247
omp_destroy_lock, 247
omp_get_num_threads, 214, 215, 260
omp_get_thread_num, 214, 215, 260
omp_get_wtime function, 64
omp_init_lock, 247
omp_lock_t, 247, 248
OMP_SCHEDULE, 240
omp_set_lock, 248, 316
omp_test_lock, 338
omp_unset_lock, 248
One_sided communication, 55, 75
OpenCL, 355
OpenMP, 8–9, 316, 318, 337–338
  compiling and running, 211–212
  directives-based shared-memory, 210
  error checking, 215–216
  evaluation, 288–289, 288*t*
  forking and joining threads, 213
  loops
    bubble sort, 232–233
    for directive, 343
    odd-even transposition sort, 233–236
    scheduling, 236–241
  num_threads clause, 214
  parallel directive, 213, 214
  parallel for directive

OpenMP (*continued*)
caveats, 225–227
data dependences, 227–228
estimating π, 229–231
loop-carried dependences, 228–229
parallelizing, *n*-body solvers, 281, 284, 288–289
performance of, 318–319, 319*t*
pragmas, 210–211
and Pthreads, 209
`reduction` clause, 221–224
scope of variables, 220–221
shared-memory system, 209, 210
structured block, 213
thread of execution, 213
trapezoidal rule
critical section, 218
Foster's methodology, 216
`Trap` function, 218–220
tree search, implementation of
MPI and dynamic partitioning, 327
parallelizing, 316–318
OpenMP Architecture Review Board, 354
Operating system (OS)
multitasking, 17–18
processes, 17–18, 18*f*
Output, 56–57, 97–100
Overhead, 55, 58, 189, 241, 280

P

Page (virtual memory), 24, 72
Page fault, 25
Parallel computing, 9–10, 12
history of, 355
`parallel` directive, 213, 214
`parallel for` directive
caveats, 225–227
data dependences, 227–228
estimating π, 229–231
loop-carried dependences, 228–229
Parallel hardware, 73–74, 354
cache coherence, 43–46
interconnection networks, 35–42
MIMD systems, 32–35
shared-memory *vs.* distributed-memory, 46
SIMD systems, 29–32
Parallel odd-even transposition sort algorithm, 129–132, 131*t*
`Merge_low` function in, 136
run-times of, 135*t*
Parallel programs, 1, 10–12
design, 65–70, 76
performance, 75–76
Amdahl's law, 61–62
scalability, 62
speedup and efficiency, 58–61, 59*t*, 60*f*
timings, 63–65
running, 70
writing, 3–8, 11, 70
Parallel software, 74–75
caveats, 47–48
coordinating processes/threads, 48–49
distributed-memory programs, 53–56
programming hybrid systems, 56
shared-memory programs, 49–53
Parallel sorting algorithm, 127
Parallel systems, building, 3
Parallelization, 8, 9, 48, 61
Partial sums, computation of, 7
Partitioned global address space (PGAS) languages, 55–56
Partitioning data, 66
Partitioning loop iterations, 290
Performance evaluation, 15
results of timing, 122–125
scalability, 126–127
speedup and efficiency, 125–126
taking timings, 119–122
Ping-pong communication, 148
Pipelining, 25–28, 72
Point-to-point communications, 104, 137
collective *vs.*, 105–106
Polling, 55
Pop (stack), 303, 304
Posix Threads, *see* Pthreads
POSIX®, 153, 174, 181, 354
Prefix sums, 142
Preprocessor, 121, 259
Pragmas, 210–211
Processes, operating systems, 17–18, 18*f*
Producer threads, 242
Producer-consumer synchronization, 171–176
Program, compilation and execution, 84–86
Program counter, 16
Progress, 9
Protect (critical section), 245, 353, 354
Protein folding, 2

Pseudocode for Pthreads Terminated function, 312
pthread.h, 155, 156, 198
pthread_attr_t, 156
pthread_barrier_destroy, 203
pthread_barrier_init, 203
pthread_barrier_t, 203
pthread_barrier_wait, 203
pthread_barrierattr_t, 203
pthread_cond_broadcast, 180
pthread_cond_destroy, 181
pthread_cond_init, 181
pthread_cond_signal, 180
pthread_cond_t, 179, 180
pthread_cond_wait, 181, 311, 313, 317
  implementing, 180
pthread_condattr_t, 181
pthread_create, 156, 157
pthread_join, 158
pthread_mutex_destroy, 169
PTHREAD_MUTEX_ERRORCHECK, 354
pthread_mutex_init, 169, 203
pthread_mutex_lock, 169, 170, 173, 181, 354
PTHREAD_MUTEX_RECURSIVE, 354
pthread_mutex_t, 169
pthread_mutex_trylock, 314, 337–338, 354
pthread_mutex_unlock, 169
pthread_mutexattr_t, 169
pthread_rwlock_destroy, 188
pthread_rwlock_init, 188
pthread_rwlock_rdlock, 187
pthread_rwlock_t, 188
pthread_rwlock_unlock, 187
pthread_rwlock_wrlock, 187
pthread_rwlockattr_t, 188
pthread_t, 156, 157
Pthreads, 8–9, 212, 320, 327, 353, 354
  barriers, 181
  dynamic parallelization of tree search, 310–315
  functions, syntax, 187
  implementation of tree-search, pseudocode for, 309
  matrix-vector multiplication, 192
  and OpenMP, 337–338
  parallelizing *n*-body solvers using, 289–290
  program
    error checking, 158–159
    execution, 153–155
    preliminaries, 155–156
  read-write locks, 187–188
  running, 157–158
  for shared-memory programming, 209
  splitting stack in parallel tree-search, 314–315
  starting threads, 156–157, 159
  static parallelization of tree search, 309–310
  stopping threads, 158
  termination of tree-search, 311–314
  tree search programs
    evaluating, 315
    run-times of, 315*t*
pthread_t, 156–158
Push (stack), 303

Q

qsort (C library function), 130
Queues, 241–242

R

Race conditions, 51, 74, 165, 260
Read (of memory, cache), 20
Read access, 309
Read-lock function, 187
Read miss, 193, 254
Read-write locks, 181, 199
  implementations, 190
  performance of, 188–190
  linked list functions, 181–183
  multithreaded linked list, 183–187
  Pthreads, 187–188
Receive_work function in MPI tree search, 334
Receiving messages, 243–244
Recursive depth-first search, 302–303
Reduction clause, 221–224
Reduction operator, 223, 262
Reentrant, 197, 258
Registers, 16
Relinquish (lock, mutex), 51, 169, 181, 247, 337
Remote memory access, 55
Request (MPI nonblocking communication), 347
Ring architecture, 293
Ring pass, 292
  computation of forces, 294, 295*t*
  positions, 293, 294*f*
Row major order, 22
Run-time of parallel programs, 63
runtime schedule types, 239–240

S

Safe MPI communication, 134*f*
Safety (thread), 52–53, 195–198, 256–259
Scalability, 31, 62
Scatter (MPI communication), 110–112
Scatter-gather (in vector processors), 31
`schedule` clause, 237–238
Scheduling loops
  `auto`, 238, 261
  `dynamic` and `guided` schedule types, 239
  `runtime` schedule types, 239–240
  `schedule` clause, 237–238
  `static` schedule type, 238–239
Scope of variables, 220–221
`sem_destroy`, 175n
`sem_open`, 175
`sem_init`, 175
`sem_post`, 174, 178, 179, 199
`sem_t`, 175
`sem_wait`, 174, 178, 179, 199
`semaphore.h`, 199
Semaphores, 52, 171–179, 199, 353, 354
  functions, syntax, 175
Sending messages, 243
`Send_rejects` function in MPI tree search, 327, 331
Serial implementations
  data structures for, 305–306
  performance of, 306, 306*t*
Serial programs, 1, 66–68
  parallelizing, 68–70
  writing, 11
Serial systems, 71–73
Serialize, 58, 185, 189, 195
Shader functions, 32
Shared-memory, *see also* Distributed-memory
  interconnects, 35–37
  programs, 49–53, 151, 355
    dynamic thread, 49
    nondeterminism in, 49–52
    static thread, 49
    thread safety, 52
  systems, 8, 9*f*, 12, 33–34, 83, 84*f*, 152*f*
    with cores and caches, 43*f*
    for programming, 151, 209–210
  vs. distributed-memory, 46
SIMD systems, *see* Single instruction, multiple data systems
Simultaneous multithreading (SMT), 29
Single instruction, multiple data (SIMD) systems, 29–30
  graphics processing units, 32
  vector processors, 30–32
Single instruction, single data (SISD) system, 29
Single program, multiple data (SPMD) programs, 47–48, 88
Single-core system, 3
SISD system, *see* Single instruction, single data system
Snooping cache coherence, 44
Sorting algorithm
  parallel, 127
  serial, 127–129
Speculation, 27–28
Spin, 166
`Split_stack` function in parallel tree search, 329
SPMD programs, *see* Single program, multiple data programs
`sprintf` (C library function), 53
Stack, 303, 305, 309, 311, 314–315, 329–331
Stall, 20
Static parallelization of tree search using pthreads, 309–310
`static` storage class in C, 197, 258
`static` schedule type, 238–239
Static threads, 49
`status_p` argument, 92–93
Store (as in load/store), 31
Strided memory access, 31
`strtok` function, 196, 197, 258, 259
`strtok_r` (C library function), 197, 258
`strtol` function, 155, 213
Structured block, 213
Swap space, 24
Switched interconnects, 35
Synchronization, 7

T

Tag (MPI message tag), 90, 91
Task (Foster's methodology), 81
Task-parallelism, 6–7, 48
`Terminated` function in parallel tree search, 311–314, 317, 318, 327
Termination detection, 244

Thread-level parallelism (TLP), 28
Threads, 18, 18*f*, 48–49, 151
  of control, 152–153
  dynamic, 49
  of execution, 213
  function, 196
    for computing π, 163
  running, 157–158
    incorrect programs, 198
    `strtok` function, 258, 259
    `Tokenize` function, 257
  starting, 156–157
    approaches to, 159
  static, 49
  stopping, 158
Thread-safety, 52–53, 195–198
`timer.h`, 121, 138
Timing parallel programs, 64
TLP, *see* Thread-level parallelism
`Tokenize` function, 257
Toroidal mesh, 37, 74
Tour (traveling salesperson problem), 299
$T_{overhead}$, 59, 79, 123
$T_{parallel}$, 59, 76, 79, 123, 170, 253
Transactional memory, 52
Translation programs, 3
Translation-lookaside buffer (TLB), 25
`Trap` function, 218–220
  in trapezoidal rule MPI, 99*f*
Trapezoidal rule, 94–95, 95*f*
  critical section, 218
  Foster's methodology, 216
  MPI, first version of, 98*f*
  parallelizing, 96–97
  tasks and communications for, 96*f*
Travelling salesperson problem, 299
Tree search
  depth-first search, 301
  dynamic partitioning
    checking for and receiving new best tours, 333–334
    distributed termination detection, 331–333, 332*t*
    in Pthreads and in OpenMP, 327
    sending requests, 333
    splitting stack, 329–331
  directed graph, 300
  MPI, 319–326, 327–333
    performance of, 334–335, 334*t*
  nonrecursive depth-first search, 303–305
  parallelizing
    best tour data structure, 307–308
    dynamic mapping of tasks, 308
    mapping details, 307
  in Pthreads and OpenMP, 334, 337
  recursive depth-first search, 302–303
  serial implementations
    data structures for, 305–306
    performance of, 306, 306*t*
  static partitioning
    maintaining best tour, 321–325
    printing best tour, 325–326
    in Ptherads and OpernMP, 319–321
    unreceived messages, 326
Tree-structured broadcast, 108*f*
Tree-structured communication, 102–103
T_{serial}, 58, 59, 76, 79, 192
Typographical conventions, 11

U

Uniform memory access (UMA) system, 34, 34*f*
Unblock, 179, 180
Unlock, 51, 178, 190
Unpack, 145, 334
Unsafe communication in MPI, 132, 139, 140
`Update_best_tour` in parallel tree search, 316
  pseudocode for, 310

V

Valgrind, 193, 253, 265
Variable, condition, 179–181, 199
Vector addition
  parallel implementation of, 110
  serial implementation of, 109
Vector processors, 30–32
Virtual address, 24, 24*t*
Virtual memory, 23–25
`volatile` storage class, 318
  von Neumann architecture, 15–17, 16*f*
  modifications to, 18
    caching, 19–23
    hardware multithreading, 28–29

volatile storage class (*continued*)
instruction-level parallelism, 25–28
virtual memory, 23–25
von Neumann bottleneck, 16–17

W

Wait, condition in Pthreads, 207, 317, 327
Waiting for nonblocking communications in MPI, 340
Wildcard arguments, 92
Windows, 239, 240
Wrapper script for C compiler, 85
Write (to cache, memory, disk), 20
Write-back cache, 20, 44
Write miss, 193, 253
Write-through cache, 20, 44

Z

Zero-length message, 333