Foundations of Multithreaded, Parallel, and Distributed Programming

Gregory R. Andrews

Amazon
Exhibit 1011
Page 001

Foundations of Multithreaded, Parallel, and Distributed Programming

Amazon
Exhibit 1011
Page 002

Foundations of Multithreaded, Parallel, and Distributed Programming

Gregory R. Andrews
University of Arizona

Addison-Wesley

An imprint of Addison Wesley Longman, Inc.

Reading, Massachusetts • Menlo Park, California •
New York • Harlow, England • Don Mills, Ontario •
Sydney • Mexico City • Madrid • Amsterdam

Amazon
Exhibit 1011
Page 003

Acquisions Editor: *Maite Suarez-Rivas*
Associate Editor: *Jason Miranda*
Project Manager: *Trillium Project Management*
Production Assistant: *Tracy Treeful*
Copyeditor: *Stephanie Magean*
Proofreader: *Trillium Project Management*
Cover Designer: *Night & Day Design*
Composition: *Chiron, Inc.*

Library of Congress Cataloging-in-Publication Data

Andrews, Gregory R.
Foundations of multithreaded, parallel, and distributed programming / Gregory R. Andrews.
p. cm.
Includes bibliographical references.
ISBN 0-201-35752-6
1. Parallel programming (Computer science) 2. Electronic data processing—Distributed processing. 3. Threads (Computer programs) I. Title: Multithreaded, parallel, and distributed programming. II. Title.

QA76.58.A57 2000
004′.3--dc21 99-045285

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and Addison-Wesley was aware of a trademark claim, the designations have been printed in initial caps or all caps.

The programs and applications presented in this book have been included for their instructional value. They have been tested with care but are not guaranteed for any purpose. The publisher and author do not offer any warranties or representations, nor do they accept any liabilities with respect to the programs or applications.

Access the latest information about Addison-Wesley titles from our World Wide Web site: www.awlonline.com

Copyright © 2000 by Addison Wesley Longman, Inc.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the publisher. Printed in the United States of America.

1 2 3 4 5 6 7 8 9 10 CRS 0302010099

Preface

Concurrent programming originated in 1962 with the invention of channels, which are independent device controllers that make it possible to have a CPU execute a new application program at the same time that I/O operations are being executed on behalf of other, suspended application programs. Hence, concurrent programming—the word *concurrent* means happening at the same time—was initially of concern to operating systems designers. In the late 1960s, hardware designers developed multiple processor machines. This presented not only a challenge for operating systems designers but also an opportunity that application programmers could exploit.

The first major concurrent programming challenge was to solve what is now called the critical section problem. It and related problems (dining philosophers, readers/writers, etc.) led to a spate of papers in the 1960s. To harness the challenge, people developed synchronization primitives such as semaphores and monitors to simplify the programmer's task. By the mid-1970s, people came to appreciate the necessity of using formal methods to help control the inherent complexity of concurrent programs.

Computer networks were introduced in the late 1970s and early 1980s. The Arpanet supported wide-area computing, and the Ethernet established local-area networks. Networks gave rise to distributed programming, which was a major topic of the 1980s and became even more important in the 1990s. The essence of distributed programming is that processes interact by means of message passing rather than by reading and writing shared variables.

Now, at the dawn of a new century, we have seen the emergence of massively parallel processing—in which tens, hundreds, or even thousands of processors are used to solve a single problem. We have also seen the emergence of

Amazon
Exhibit 1011
Page 005

client/server computing, the Internet, and the World Wide Web. Finally, we are beginning to see multiprocessor workstations and PCs. Concurrent hardware is more prevalent than ever, and concurrent programming is more relevant than ever.

This is my third book, another attempt to capture a part of the history of concurrent programming. My first book—*Concurrent Programming: Principles and Practice*, published in 1991—gives a broad, reference-level coverage of the period between 1960 and 1990. Because new problems, programming mechanisms, and formal methods were significant topics in those decades, the book focuses on them.

My second book—*The SR Programming Language: Concurrency in Practice*, published in 1993—summarizes my work with Ron Olsson in the late 1980s and early 1990s on a specific language that can be used to write concurrent programs for both shared- and distributed-memory machines. The SR book is pragmatic rather than formal, showing how to solve numerous problems in a single programming language.

This book is an outgrowth of my past and a reflection of what I think is important now and will be in the future. I have drawn heavily from material in the *Concurrent Programming* book, but I have completely rewritten every section that I have retained and have rewritten examples to use pseudo-C rather than SR. I have added new material throughout, especially on parallel scientific programming. I have also included case studies of the most important languages and software libraries, with complete sample programs. Finally, I have a new vision for the role of this book—in classrooms and in personal libraries.

A New Vision and Role

Parallel and distributed computing are today pervasive concepts. As usual in computer science, advances have been led by hardware designers, who keep building bigger, faster, more powerful computers and communication networks. For the most part, they are succeeding—witness the stock market!

New computers and networks create new challenges and opportunities, and for once software designers are no longer all that far behind. Consider Web browsers, Internet commerce, Pthreads, Java, MPI, and so on—and again, witness the stock market! These software products are specifically designed to take advantage of concurrency in hardware and applications. In short, much of the computing world is now concurrent!

Aspects of concurrent computing—whether multithreaded, parallel, or distributed—are now covered in almost every advanced course in computer science.

Amazon
Exhibit 1011
Page 006

Reflecting the history of the topic, operating systems courses lead the way—covering topics like multithreading, communication protocols, and distributed file systems. Architecture courses cover multiprocessors and networks. Compilers courses cover compilation issues for parallel machines. Theory courses cover models for parallel computing. Algorithms courses cover parallel algorithms. Database courses cover locking and distributed databases. Graphics courses make use of parallelism for rendering and ray tracing. The list goes on. In addition, concurrent computing has become a fundamental tool in a wide range of science and engineering disciplines.

The main purpose of this book—as reflected in the title—is to lay the foundations for programming multithreaded, parallel, and distributed computations. The specific goal is to provide a text that can be used for, or to create, an advanced undergraduate/beginning graduate course. Whenever a topic in computing has become pervasive, as concurrency surely has, we have added foundation courses to provide students with basic knowledge of a topic. Similarly, whenever a topic has become well understood, as concurrency now is, we have migrated the topic to the core curriculum.

I have tried to cover those aspects of parallel and distributed computing that I think every computer science student should know. This includes basic principles, programming techniques, major applications, implementations, and performance issues. In addition, I have included case studies of many of the most important languages and software libraries: Pthreads (three chapters), Java (three chapters), CSP, Linda, MPI (two chapters), Ada, SR, and OpenMP. Each case study describes relevant parts of the language or library and then presents a complete sample program. (Sources for the programs are available on this book's Web site. In addition, I summarize several additional languages, models, and tools for parallel scientific computing in Chapter 12.

On the other hand, no single book can cover everything—and still be affordable—so students and instructors may wish to augment this text with others. The Historical Notes and the References at the end of each chapter describe additional material and provide pointers for further study. This book's Web pages provide further information and links to relevant material.

Content Overview

This book contains 12 chapters. Chapter 1 summarizes basic concepts of concurrency, hardware, and applications. Then it examines five representative applications: matrix multiplication, adaptive quadrature, Unix pipes, file systems, and distributed matrix multiplication. Each application is simple yet useful; together

Amazon
Exhibit 1011
Page 007

they illustrate the range of issues and five recurring programming styles. The last section of Chapter 1 summarizes the programming notation that is used in the text.

The remaining chapters are organized into three parts. Part 1 describes concurrent programming mechanisms that use shared variables, and hence that are directly suitable for shared-memory machines. Chapter 2 introduces fundamental concepts of processes and synchronization; the chapter uses a series of examples to illustrate the key points and ends with a discussion of the formal semantics of concurrency. Understanding the semantic concepts will help you understand parts of later chapters, but it should also be sufficient to refer back to them when necessary. Chapter 3 shows how to implement and use locks and barriers; it also describes data parallel algorithms and a parallel programming technique called a bag of tasks. Chapter 4 describes semaphores and gives numerous examples of how to use them. Semaphores were the first high-level concurrent programming mechanism and remain one of the most important. Chapter 5 covers monitors in detail. Monitors were introduced in a seminal 1974 paper, somewhat lost favor in the 1980s and early 1990s, but have regained importance with the Java language. Finally, Chapter 6 describes how to implement processes, semaphores, and monitors on both uniprocessors and multiprocessors.

Part 2 covers distributed programming, in which processes communicate and synchronize by means of messages. Chapter 7 describes message passing using `send` and `receive` primitives. It shows how to use these primitives to program filters (which have one-way communication), clients and servers (which have two-way communication), and interacting peers (which have back and forth communication). Chapter 8 examines two additional communication primitives: remote procedure call (RPC) and rendezvous. With these, a client process initiates a communication by issuing a `call`—which is implicitly a `send` followed by a `receive`; the communication is serviced either by a new process (RPC) or by a rendezvous with an existing process. Chapter 9 describes several paradigms for process interaction in distributed programs. These include three that are commonly used in parallel computations—manager/workers, heartbeat, and pipeline—as well as four that arise in distributed systems—probe/echo, broadcast, token passing, and replicated servers. Finally, Chapter 10 describes how to implement message passing, RPC, and rendezvous. That chapter also shows how to implement what is called a distributed shared memory, which supports a shared-memory programming model in a distributed environment.

Part 3 covers parallel programming, especially for high-performance scientific computations. (Many other kinds of parallel computations are described in earlier chapters and in the exercises of several chapters.) The goal of a parallel program is speedup—namely, to solve a problem faster by using multiple

Amazon
Exhibit 1011
Page 008

processors. Parallel programs are written using shared variables or message passing; hence they employ the techniques described in Parts 1 and 2. Chapter 11 examines the three major classes of scientific computing applications: grid, particle, and matrix computations. These arise in simulating (modeling) physical and biological systems; matrix computations are also used for such things as economic forecasting. Chapter 12 surveys the most important tools that are used to write parallel scientific computations: libraries (Pthreads, MPI, and OpenMP), parallelizing compilers, languages and models, and higher-level tools such as metacomputations.

The end of each chapter provides historical notes, references, and an extensive set of exercises. The historical notes summarize the origin and evolution of each topic and how the topics relate to each other. The notes also describe the papers and books listed in the reference section. The exercises explore the topics covered in each chapter and also introduce additional applications.

Classroom Use

I use this book each spring at the University of Arizona for a class of about 60 students. About half the students are undergraduates and half are graduates. Most are computer science majors; the class is an elective for them, but most majors end up taking it. The class is also taken each year by a few Ph.D. students from other departments who have an interest in computational science and parallel computing. Most of the undergraduates are simultaneously taking our undergraduate operating systems course; the CS graduate students have all had such a course and many have also taken our advanced operating systems course.

Our operating systems classes, like those elsewhere, examine the synchronization *needs* of operating systems, and explore how processes, synchronization, and other OS topics are implemented, mostly on single processor machines. My class shows how to *use* concurrency as a general programming tool in a broad range of applications. I look at programming techniques, higher-level concepts, parallel computing, and distributed computing. In a sense, my class is to an OS class somewhat like what a comparative programming languages class is to a compilers class.

In the first half of my class, I cover Chapter 2 in some detail but then go fairly rapidly through the other chapters in Part I, emphasizing topics not covered in the OS classes: the Test-and-Test-and-Set protocol, the Ticket Algorithm, barriers, passing the baton for semaphores, some of the monitor programming techniques, and the multiprocessor kernel. In addition, students are introduced to the Pthreads library and Java's threads and synchronized methods in homework

Amazon
Exhibit 1011
Page 009

assignments. I also include a parallel computing project after my lectures on barriers (drawing material from Chapter 11).

In the second half of my class I cover much of Part 2 of the book on distributed programming. We look at message passing, and how it is used to program both distributed systems and distributed parallel computations. Then we look at RPC and rendezvous, their realization in Java and Ada, and distributed systems applications. Finally, we look at each of the paradigms in Chapter 9, again using applications from parallel and distributed systems to motivate and illustrate them. Students are introduced to the MPI library in a homework assignment and again use Java.

During the term I give four homework assignments, two in-class exams, a parallel computing project, and a final project. (See the book's Web site for samples.) Each homework assignment consists of written problems from the exercises, as well as one or two programming problems. Graduate students are required to do all problems; undergraduates do two-thirds of the problems (of their choice). Exams are administered similarly: graduates do all problems, undergraduates choose a subset of the problems to answer. At Arizona we use SR in the earliest programming assignments, and students have the option of using SR for many of the later assignments, but they are encouraged to use Pthreads, Java, and MPI.

The parallel programming project addresses one of the problems in Chapter 11 (often a grid computation). Students write programs and do timing experiments on a small shared-memory multiprocessor. Graduate students implement harder algorithms and are also required to write a fairly detailed report on their experiments. The final project is a paper or programming project on some aspect of distributed computing. Students get to choose what they want to do, subject to my approval. Most students do implementation projects; many work in pairs. The students produce an amazing variety of interesting systems, and most include a graphical user interface.

Even though the students in my class have a wide range of backgrounds, almost all give the class an excellent evaluation. Undergraduates frequently remark on how well it meshes with their OS class; they like seeing concurrency from another perspective, learning about multiprocessors, and seeing a broad range of applications and tools. Graduate students remark on how the class ties together a variety of things they had heard something about, and also introduces them to the breadth of concurrent programming; many of them would, however, like to see more coverage of parallel computing. Eventually we hope to be able to offer separate classes for undergraduates and graduates. I would not change much for the undergraduate-only class, but I would cover slightly fewer topics in somewhat more depth. For the graduate version I would spend less time on

Amazon
Exhibit 1011
Page 010

material they were familiar with (Part 1) and more time on parallel applications and parallel computing tools (Part 3).

This book is ideally suited for classes that cover the range of concurrent programming mechanisms, tools, and applications. Almost all students will make use of *some* of what is covered here, but not many will do just parallel computing, just distributed systems, or just programming in Java. However, the book can also be used as a supplement to classes with a more specialized focus. For example, a parallel computing class might use this book together with one that covers parallel algorithms and applications.

Online Resources

The "home page" for the book is located at

```
http://www.cs.arizona.edu/people/greg/mpdbook
```

The Web site contains the source for the larger programs in the text, links to software and information on the languages and libraries described in the case studies, and a variety of other useful links. It also contains extensive information from my Arizona class, including the syllabus, lecture notes, and copies of homework assignments, projects, and exams. Information on this book is also available at

```
http://www.awlonline.com/cs
```

Despite my best efforts, this book no doubt contains errors, so there will be (too soon I'm sure) an errata page. Contact your Addison Wesley representative for information on obtaining a copy of the Instructor's Manual for this book. Other material will no doubt be added in the future.

Acknowledgments

I received many useful suggestions from the reviewers of earlier drafts of this book. Martti Tienari and his students at the University of Helsinki found several subtle errors. My most recent Ph.D. students, Vince Freeh and Dave Lowenthal, provided comments on new chapters and helped debug my thinking, if not my programs, over the past several years. The students in my spring 1999 class served as guinea pigs and found several embarrassing errors. Andy Bernat also provided feedback from his spring 1999 class at the University of Texas, El Paso. Thank you to the following reviewers for their invaluable feedback: Jaspal Subhlok (Carnegie Mellon University), Boleslaw Szymanski (Rensselaer

Amazon
Exhibit 1011
Page 011

Polytechnic Institute), G.S. Stiles (Utah State University), Narsingh Deo (University of Central Florida), Janet Hartman (Illinois State University), Nan C. Schaller (Rochester Institute of Technology), and Mark Fienup (University of Northern Iowa).

The National Science Foundation has supported my research for many years, most recently through grants CCR-9415303 and ACR-9720738. A research infrastructure grant from the NSF (CDA-9500991), helped support the equipment used to prepare the book and test the programs.

Most importantly, I want to thank my wife, Mary, who once again put up with the thousands of hours I spent working on this book—despite having vowed "never again" after finishing the SR book.

Greg Andrews
Tucson, Arizona

Amazon
Exhibit 1011
Page 012

Table of Contents

Preface v

Chapter 1: The Concurrent Computing Landscape . 1

1.1 The Essence of Concurrent Programming 2
1.2 Hardware Architectures 4
1.2.1 Processors and Caches 4
1.2.2 Shared-Memory Multiprocessors 6
1.2.3 Distributed-Memory Multicomputers and Networks 8
1.3 Applications and Programming Styles 10
1.4 Iterative Parallelism: Matrix Multiplication 13
1.5 Recursive Parallelism: Adaptive Quadrature 17
1.6 Producers and Consumers: Unix Pipes 19
1.7 Clients and Servers: File Systems 21
1.8 Peers: Distributed Matrix Multiplication 23
1.9 Summary of Programming Notation 26
1.9.1 Declarations 26
1.9.2 Sequential Statements 27
1.9.3 Concurrent Statements, Processes, and Procedures 29
1.9.4 Comments 31
Historical Notes 31
References 33
Exercises 34

Amazon
Exhibit 1011
Page 013

Part 1: Shared-Variable Programming 39

Chapter 2: Processes and Synchronization 41

2.1 States, Actions, Histories, and Properties 42
2.2 Parallelization: Finding Patterns in a File 44
2.3 Synchronization: The Maximum of an Array 48
2.4 Atomic Actions and Await Statements 51
2.4.1 Fine-Grained Atomicity 51
2.4.2 Specifying Synchronization: The Await Statement 54
2.5 Producer/Consumer Synchronization 56
2.6 A Synopsis of Axiomatic Semantics 57
2.6.1 Formal Logical Systems 58
2.6.2 A Programming Logic 59
2.6.3 Semantics of Concurrent Execution 62
2.7 Techniques for Avoiding Interference 65
2.7.1 Disjoint Variables 65
2.7.2 Weakened Assertions 66
2.7.3 Global Invariants 68
2.7.4 Synchronization 69
2.7.5 An Example: The Array Copy Problem Revisited 70
2.8 Safety and Liveness Properties 72
2.8.1 Proving Safety Properties 73
2.8.2 Scheduling Policies and Fairness 74
Historical Notes 77
References 80
Exercises 81

Chapter 3: Locks and Barriers 93

3.1 The Critical Section Problem 94
3.2 Critical Sections: Spin Locks 97
3.2.1 Test and Set 98
3.2.2 Test and Test and Set 100
3.2.3 Implementing Await Statements 101
3.3 Critical Sections: Fair Solutions 104
3.3.1 The Tie-Breaker Algorithm 104
3.3.2 The Ticket Algorithm 108
3.3.3 The Bakery Algorithm 111
3.4 Barrier Synchronization 115
3.4.1 Shared Counter 116
3.4.2 Flags and Coordinators 117

Amazon
Exhibit 1011
Page 014

3.4.3 Symmetric Barriers 120
3.5 Data Parallel Algorithms 124
3.5.1 Parallel Prefix Computations 124
3.5.2 Operations on Linked Lists 127
3.5.3 Grid Computations: Jacobi Iteration 129
3.5.4 Synchronous Multiprocessors 131
3.6 Parallel Computing with a Bag of Tasks 132
3.6.1 Matrix Multiplication 133
3.6.2 Adaptive Quadrature 134
Historical Notes 135
References 139
Exercises 141

Chapter 4: Semaphores 153
4.1 Syntax and Semantics 154
4.2 Basic Problems and Techniques 156
4.2.1 Critical Sections: Mutual Exclusion 156
4.2.2 Barriers: Signaling Events 156
4.2.3 Producers and Consumers: Split Binary Semaphores 158
4.2.4 Bounded Buffers: Resource Counting 160
4.3 The Dining Philosophers 164
4.4 Readers and Writers 166
4.4.1 Readers/Writers as an Exclusion Problem 167
4.4.2 Readers/Writers Using Condition Synchronization 169
4.4.3 The Technique of Passing the Baton 171
4.4.4 Alternative Scheduling Policies 175
4.5 Resource Allocation and Scheduling 178
4.5.1 Problem Definition and General Solution Pattern 178
4.5.2 Shortest-Job-Next Allocation 180
4.6 Case Study: Pthreads 184
4.6.1 Thread Creation 185
4.6.2 Semaphores 186
4.6.3 Example: A Simple Producer and Consumer 186
Historical Notes 188
References 190
Exercises 191

Chapter 5: Monitors 203
5.1 Syntax and Semantics 204
5.1.1 Mutual Exclusion 206

Amazon
Exhibit 1011
Page 015

5.1.2 Condition Variables 207
5.1.3 Signaling Disciplines 208
5.1.4 Additional Operations on Condition Variables 212
5.2 Synchronization Techniques 213
5.2.1 Bounded Buffers: Basic Condition Synchronization 213
5.2.2 Readers and Writers: Broadcast Signal 215
5.2.3 Shortest-Job-Next Allocation: Priority Wait 217
5.2.4 Interval Timer: Covering Conditions 218
5.2.5 The Sleeping Barber: Rendezvous 221
5.3 Disk Scheduling: Program Structures 224
5.3.1 Using a Separate Monitor 228
5.3.2 Using an Intermediary 230
5.3.3 Using a Nested Monitor 235
5.4 Case Study: Java 237
5.4.1 The Threads Class 238
5.4.2 Synchronized Methods 239
5.4.3 Parallel Readers/Writers 241
5.4.4 Exclusive Readers/Writers 243
5.4.5 True Readers/Writers 245
5.5 Case Study: Pthreads 246
5.5.1 Locks and Condition Variables 246
5.5.2 Example: Summing the Elements of a Matrix 248
Historical Notes 250
References 253
Exercises 255

Chapter 6: Implementations 265
6.1 A Single-Processor Kernel 266
6.2 A Multiprocessor Kernel 270
6.3 Implementing Semaphores in a Kernel 276
6.4 Implementing Monitors in a Kernel 279
6.5 Implementing Monitors Using Semaphores 283
Historical Notes 284
References 286
Exercises 287

Part 2: Distributed Programming 291

Chapter 7: Message Passing 295
7.1 Asynchronous Message Passing 296

Amazon
Exhibit 1011
Page 016

7.2 Filters: A Sorting Network 298
7.3 Clients and Servers 302
7.3.1 Active Monitors 302
7.3.2 A Self-Scheduling Disk Server 308
7.3.3 File Servers: Conversational Continuity 311
7.4 Interacting Peers: Exchanging Values 314
7.5 Synchronous Message Passing 318
7.6 Case Study: CSP 320
7.6.1 Communication Statements 321
7.6.2 Guarded Communication 323
7.6.3 Example: The Sieve of Eratosthenes 326
7.6.4 Occam and Modern CSP 328
7.7 Case Study: Linda 334
7.7.1 Tuple Space and Process Interaction 334
7.7.2 Example: Prime Numbers with a Bag of Tasks 337
7.8 Case Study: MPI 340
7.8.1 Basic Functions 341
7.8.2 Global Communication and Synchronization 343
7.9 Case Study: Java 344
7.9.1 Networks and Sockets 344
7.9.2 Example: A Remote File Reader 345
Historical Notes 348
References 351
Exercises 353

Chapter 8: RPC and Rendezvous 361
8.1 Remote Procedure Call 362
8.1.1 Synchronization in Modules 364
8.1.2 A Time Server 365
8.1.3 Caches in a Distributed File System 367
8.1.4 A Sorting Network of Merge Filters 370
8.1.5 Interacting Peers: Exchanging Values 371
8.2 Rendezvous 373
8.2.1 Input Statements 374
8.2.2 Client/Server Examples 376
8.2.3 A Sorting Network of Merge Filters 379
8.2.4 Interacting Peers: Exchanging Values 381
8.3 A Multiple Primitives Notation 382
8.3.1 Invoking and Servicing Operations 382
8.3.2 Examples 384

Amazon
Exhibit 1011
Page 017

8.4 Readers/Writers Revisited 386
8.4.1 Encapsulated Access 387
8.4.2 Replicated Files 389
8.5 Case Study: Java 393
8.5.1 Remote Method Invocation 393
8.5.2 Example: A Remote Database 395
8.6 Case Study: Ada 397
8.6.1 Tasks 398
8.6.2 Rendezvous 399
8.6.3 Protected Types 401
8.6.4 Example: The Dining Philosophers 403
8.7 Case Study: SR 406
8.7.1 Resources and Globals 406
8.7.2 Communication and Synchronization 408
8.7.3 Example: Critical Section Simulation 409
Historical Notes 411
References 415
Exercises 416

Chapter 9: Paradigms for Process Interaction 423
9.1 Manager/Workers (Distributed Bag of Tasks) 424
9.1.1 Sparse Matrix Multiplication 424
9.1.2 Adaptive Quadrature Revisited 428
9.2 Heartbeat Algorithms 430
9.2.1 Image Processing: Region Labeling 432
9.2.2 Cellular Automata: The Game of Life 435
9.3 Pipeline Algorithms 437
9.3.1 A Distributed Matrix Multiplication Pipeline 438
9.3.2 Matrix Multiplication by Blocks 441
9.4 Probe/Echo Algorithms 444
9.4.1 Broadcast in a Network 444
9.4.2 Computing the Topology of a Network 448
9.5 Broadcast Algorithms 451
9.5.1 Logical Clocks and Event Ordering 452
9.5.2 Distributed Semaphores 454
9.6 Token-Passing Algorithms 457
9.6.1 Distributed Mutual Exclusion 457
9.6.2 Termination Detection in a Ring 460
9.6.3 Termination Detection in a Graph 462
9.7 Replicated Servers 465

Amazon
Exhibit 1011
Page 018

9.7.1 Distributed Dining Philosophers 466
9.7.2 Decentralized Dining Philosophers 467
Historical Notes 471
References 474
Exercises 477

Chapter 10: Implementations 487
10.1 Asynchronous Message Passing 488
10.1.1 Shared-Memory Kernel 488
10.1.2 Distributed Kernel 491
10.2 Synchronous Message Passing 496
10.2.1 Direct Communication Using Asynchronous Messages 497
10.2.2 Guarded Communication Using a Clearinghouse 498
10.3 RPC and Rendezvous 504
10.3.1 RPC in a Kernel 504
10.3.2 Rendezvous Using Asynchronous Message Passing 507
10.3.3 Multiple Primitives in a Kernel 509
10.4 Distributed Shared Memory 515
10.4.1 Implementation Overview 516
10.4.2 Page Consistency Protocols 518
Historical Notes 520
References 521
Exercises 522

Part 3: Parallel Programming 527

Chapter 11: Scientific Computing 533
11.1 Grid Computations 534
11.1.1 Laplace's Equation 534
11.1.2 Sequential Jacobi Iteration 535
11.1.3 Jacobi Iteration Using Shared Variables 540
11.1.4 Jacobi Iteration Using Message Passing 541
11.1.5 Red/Black Successive Over-Relaxation (SOR) 546
11.1.6 Multigrid Methods 549
11.2 Particle Computations 553
11.2.1 The Gravitational N-Body Problem 554
11.2.2 Shared-Variable Program 555
11.2.3 Message-Passing Programs 559

Amazon
Exhibit 1011
Page 019

11.2.4 Approximate Methods 569
11.3 Matrix Computations 573
11.3.1 Gaussian Elimination 573
11.3.2 LU Decomposition 575
11.3.3 Shared-Variable Program 576
11.3.4 Message-Passing Program 581
Historical Notes 583
References 584
Exercises 585

Chapter 12: Languages, Compilers, Libraries, and Tools 591
12.1 Parallel Programming Libraries 592
12.1.1 Case Study: Pthreads 593
12.1.2 Case Study: MPI 593
12.1.3 Case Study: OpenMP 595
12.2 Parallelizing Compilers 603
12.2.1 Dependence Analysis 604
12.2.2 Program Transformations 607
12.3 Languages and Models 614
12.3.1 Imperative Languages 616
12.3.2 Coordination Languages 619
12.3.3 Data Parallel Languages 620
12.3.4 Functional Languages 623
12.3.5 Abstract Models 626
12.3.6 Case Study: High-Performance Fortran (HPF) 629
12.4 Parallel Programming Tools 633
12.4.1 Performance Measurement and Visualization 633
12.4.2 Metacomputers and Metacomputing 634
12.4.3 Case Study: The Globus Toolkit 636
Historical Notes 638
References 642
Exercises 644

Glossary 647

Index 657

Amazon
Exhibit 1011
Page 020

1

The Concurrent Computing Landscape

Imagine the following scenario:

> Several cars want to drive from point A to point B. They can compete for space on the same road and end up either following each other or competing for positions (and having accidents!). Or they could drive in parallel lanes, thus arriving at about the same time without getting in each other's way. Or they could travel different routes, using separate roads.

This scenario captures the essence of concurrent computing: There are multiple tasks that need to be done (cars moving). Each can execute one at a time on a single processor (road), execute in parallel on multiple processors (lanes in a road), or execute on distributed processors (separate roads). However, tasks often need to synchronize to avoid collisions or to stop at traffic lights or stop signs.

This book is an "atlas" of concurrent computing. It examines the kinds of cars (processes), the routes they might wish to travel (applications), the patterns of roads (hardware), and the rules of the road (communication and synchronization). So load up the tank and get started.

In this chapter we describe the legend of the concurrent programming map. Section 1.1 introduces fundamental concepts. Sections 1.2 and 1.3 describe the kinds of hardware and applications that make concurrent programming both interesting and challenging. Sections 1.4 through 1.8 describe and illustrate five recurring programming styles: iterative parallelism, recursive parallelism, producers and consumers, clients and servers, and interacting peers. The last section defines the programming notation that is used in the examples and the remainder of the book.

Later chapters cover applications and programming techniques in detail. The chapters are organized into three parts—shared-variable programming,

Amazon
Exhibit 1011
Page 021

distributed (message-based) programming, and parallel programming. The introduction to each part and each chapter gives a road map, summarizing where we have been and where we are going.

1.1 The Essence of Concurrent Programming

A concurrent program contains two or more processes that work together to perform a task. Each process is a sequential program—namely, a sequence of statements that are executed one after another. Whereas a sequential program has a *single thread of control*, a concurrent program has *multiple threads of control*.

The processes in a concurrent program work together by *communicating* with each other. Communication is programmed using shared variables or message passing. When shared variables are used, one process writes into a variable that is read by another. When message passing is used, one process sends a message that is received by another.

Whatever the form of communication, processes also need to *synchronize* with each other. There are two basic kinds of synchronization: mutual exclusion and condition synchronization. *Mutual exclusion* is the problem of ensuring that *critical sections* of statements do not execute at the same time. *Condition synchronization* is the problem of delaying a process until a given condition is true. As an example, communication between a producer process and a consumer process is often implemented using a shared memory buffer. The producer writes into the buffer; the consumer reads from the buffer. Mutual exclusion is required to ensure that the producer and consumer do not access the buffer at the same time, and hence that a partially written message is not read prematurely. Condition synchronization is used to ensure that a message is not read by the consumer until after it has been written by the producer.

The history of concurrent programming has followed the same stages as other experimental areas of computer science. The topic arose due to opportunities presented by hardware developments and has evolved in response to technological changes. Over time, the initial, *ad hoc* approaches have coalesced into a collection of core principles and general programming techniques.

Concurrent programming originated in the 1960s within the context of operating systems. The motivation was the invention of hardware units called *channels* or *device controllers*. These operate independently of a controlling processor and allow an I/O operation to be carried out concurrently with continued execution of program instructions by the central processor. A channel communicates with the central processor by means of an *interrupt*—a hardware signal that says "stop what you are doing and start executing a different sequence of instructions."

Amazon
Exhibit 1011
Page 022

The programming challenge—indeed the intellectual challenge—that resulted from the introduction of channels was that now parts of a program could execute in an unpredictable order. Hence, if one part of a program is updating the value of a variable, an interrupt might occur and lead to another part of the program trying to change the value of the variable. This specific problem came to be known as the *critical section problem*, which we cover in detail in Chapter 3.

Shortly after the introduction of channels, hardware designers conceived of multiprocessor machines. For a couple decades these were too expensive to be widely used, but now all large machines have multiple processors. Indeed, the largest machines have hundreds of processors and are often called *massively parallel processors*, or MPPs. Even most personal computers will soon have a few processors.

Multiprocessor machines allow different application programs to execute at the same time on different processors. They also allow a *single* application program to execute faster if it can be rewritten to use multiple processors. But how does one synchronize the activity of concurrent processes? How can one use multiple processors to make an application run faster?

To summarize, channels and multiprocessors provide both opportunities and challenges. When writing a concurrent program one has to make decisions about what kinds of processes to employ, how many to use, and how they should interact. These decisions are affected by the application and by the underlying hardware on which the program will run. Whatever choices are made, the key to developing a *correct* program is to ensure that process interaction is properly synchronized.

This book examines all aspects of concurrent programming. However, we focus on *imperative programs* with explicit concurrency, communication, and synchronization. In particular, the programmer has to specify the actions of each process and how they communicate and synchronize. This contrasts with *declarative programs*—e.g., functional or logic programs—in which concurrency is implicit and there is no reading and writing of a program state. In declarative programs, independent parts of the program may execute in parallel; they communicate and synchronize implicitly when one part depends on the results produced by another. Although the declarative approach is interesting and important—see Chapter 12 for more information—the imperative approach is much more widely used. In addition, to implement a declarative program on a traditional machine, one has to write an imperative program.

We also focus on concurrent programs in which process execution is *asynchronous*—namely, each process executes at its own rate. Such programs can be executed by interleaving the processes on a single processor or by executing the processes in parallel on a multiple instruction stream, multiple data stream (MIMD) multiprocessor. This class of machines includes shared-memory

Amazon
Exhibit 1011
Page 023

order. In fact, there are `n!` different orders in which the array of processes could write the values (`n` factorial is the number of permutations of `n` values).

We will declare and call procedures and functions in essentially the same way as in C. Two simple examples are

```
int addOne(int v) {   # an integer function
  return (v + 1);
}

main() {   # a "void" procedure
  int n, sum;
  read(n);   # read an integer from stdin
  for [i = 1 to n]
    sum = sum + addOne(i);
  write("the final value is", sum);
}
```

If the input value `n` is `5`, this program writes the line:

```
the final value is 20
```

1.9.4 Comments

We will write comments in one of two ways. One-line comments begin with the sharp character, `#`, and terminate with the end of the line. Multiple-line comments begin with `/*` and end with `*/`. We use `#` to introduce one-line comments because the C++/Java one-line comment symbol, `//`, has long been used in concurrent programming as the separator between arms of concurrent statements.

An *assertion* is a predicate that specifies a condition that is assumed to be true at a certain point in a program. (Assertions are described in detail in Chapter 2.) Assertions can be thought of as very precise comments, so we will specify them by lines that begin with two sharp characters, as in

```
## x > 0
```

This comment asserts that `x` is positive.

Historical Notes

As noted in the text, concurrent programming originated in the 1960s after the introduction of independent device controllers (channels). Operating systems were the first software systems to be organized as multithreaded concurrent programs. The research and initial prototypes leading to modern operating systems

Amazon
Exhibit 1011
Page 024

occurred in the late 1960s and early 1970s. The first textbooks on operating systems appeared in the early 1970s.

The introduction of computer networks in the 1970s led to the development of distributed systems. The invention of the Ethernet in the late 1970s greatly accelerated the pace of activity. Almost immediately there was a plethora of new languages, algorithms, and applications. Subsequent hardware developments spurred further activity. For example, once workstations and local area networks became relatively cheap, people developed client/server computing; more recently the Internet has spawned Java, Web browsers, and a plethora of new applications.

Multiprocessors were invented in the 1970s, the most notable machines being the Illiac SIMD multiprocessors developed at the University of Illinois. Early machines were costly and specialized, however, so for many years high-performance scientific computing was done on vector processors. This started to change in the mid-1980s with the invention of hypercube machines at the California Institute of Technology and their commercial introduction by Intel. Then Thinking Machines introduced the massively parallel Connection Machine. Also, Cray Research and other makers of vector processors started to produce multiprocessor versions of their machines. For several years, numerous companies (and machines) came, and almost as quickly went. However, the set of players and machines is now fairly stable, and high-performance computing is pretty much synonymous with massively parallel processing.

The Historical Notes in subsequent chapters describe developments that are covered in the chapters themselves. Described below are several general references on hardware, operating systems, distributed systems, and parallel computing. These are the books the author has consulted most often while writing this text. (If there is a topic you wish to learn more about, you can also try a good Internet search engine.)

The now classic textbook on computer architecture is Hennessy and Patterson [1996]. If you want to learn how caches, interconnection networks, and multiprocessors work, start there. Hwang [1993] describes high-performance computer architectures. Almasi and Gottlieb [1994] give an overview of parallel computing applications, software, and architectures.

The most widely used textbooks on operating systems are Tanenbaum [1992] and Silberschatz, Peterson, and Galvin [1998]. Tanenbaum [1995] also gives an excellent overview of distributed operating systems. Mullender [1993] contains a superb collection of chapters on all aspects of distributed systems, including reliability and fault tolerance, transaction processing, file systems, real-time systems, and security. Two additional textbooks—Bacon [1998] and Bernstein and Lewis [1993]—cover multithreaded concurrent systems, including distributed database systems.

Amazon
Exhibit 1011
Page 025

machine. This makes SIMD machines attractive for large problems that can be solved using data parallel algorithms. On the other hand, SIMD machines are special purpose in the sense that the entire machine executes one program at a time. (This is the main reason they have fallen out of favor.) Moreover, it is a challenge for the programmer to keep the processors busy doing useful work. In the above algorithm, for example, fewer and fewer processors update `sum[i]` on each iteration, yet each has to evaluate the condition in the `if` statement and then, if the condition is false, has to delay until all other processes have updated `sum`. In general, the execution time of an `if` statement is the *total* of the execution time for *every* branch, even if the branch is not taken. For example, the time to execute an `if/then/else` statement on each processor is the sum of the times to evaluate the condition, execute the `then` part, and execute the `else` part.

3.6 Parallel Computing with a Bag of Tasks

Many iterative problems can be solved using a data parallel programming style, as shown in the previous section. Recursive problems that arise from the divide and conquer paradigm can be parallelized by executing recursive calls in parallel rather than sequentially, as was shown in Section 1.5.

This section presents a different way to implement parallel computations using what is called a *bag of tasks*. A task is an independent unit of work. Tasks are placed in a bag that is shared by two or more worker processes. Each worker executes the following basic code:

```
while (true) {
   get a task from the bag;
   if (no more tasks)
      break;      # exit the while loop
   execute the task, possibly generating new ones;
}
```

This approach can be used to implement recursive parallelism, in which case the tasks are the recursive calls. It can also be used to solve iterative problems having a fixed number of independent tasks.

The bag-of-tasks paradigm has several useful attributes. First, it is quite easy to use. One has only to define the representation for a task, implement the bag, program the code to execute a task, and figure out how to detect termination. Second, programs that use a bag of tasks are *scalable*, meaning that one can use any number of processors merely by varying the number of workers. (However, the performance of the program might not scale.) Finally, the paradigm makes it

Amazon
Exhibit 1011
Page 026

easy to implement *load balancing*. If tasks take different amounts of time to execute, some workers will probably execute more than others. However, as long as there are enough more tasks than workers (e.g., two or three times as many), the total amount of computation executed by each worker should be approximately the same.

Below we show how to use a bag of tasks to implement matrix multiplication and adaptive quadrature. In matrix multiplication there is a fixed number of tasks. With adaptive quadrature, tasks are created dynamically. Both examples use critical sections to protect access to the bag and use barrier-like synchronization to detect termination.

3.6.1 Matrix Multiplication

Consider again the problem of multiplying two `n` × `n` matrices `a` and `b`. This requires computing n^2 inner products, one for each combination of a row of `a` and a column of `b`. Each inner product is an independent computation, so each could be executed in parallel. However, suppose we are going to execute the program on a machine with `PR` processors. Then we would like to employ `PR` worker processes, one per processor. To balance the computational load, each should compute about the same number of inner products. In Section 1.4 we statically assigned parts of the computation to each worker. Here, we will use a bag of tasks, and have each worker grab a task whenever it is needed.

If `PR` is much smaller than `n`, a good size for a task would be one or a few rows of the result matrix `c`. (This leads to reasonable data locality for matrices `a` and `c`, assuming matrices are stored in row-major order.) For simplicity, we will use single rows. Initially, the bag contains `n` tasks, one per row. Since these can be ordered in any way we wish, we can represent the bag by simply counting rows:

```
int nextRow = 0;
```

A worker gets a task out of the bag by executing the atomic action:

```
⟨ row = nextRow; nextRow++; ⟩
```

where `row` is a local variable. The bag is empty if `row` is at least `n`. The atomic action above is another instance of drawing a ticket. It can be implemented using a Fetch-and-Add instruction, if one is available, or by using locks to protect the critical section.

Figure 3.20 contains an outline of the full program. We assume that the matrices have been initialized. Workers compute inner products in the usual way. The program terminates when each worker has broken out of its `while` loop. If

Amazon
Exhibit 1011
Page 027

```
int nextRow = 0; # the bag of tasks
double a[n,n], b[n,n], c[n,n];

process Worker[w = 1 to P] {
  int row;
  double sum;  # for inner products
  while (true) {
    # get a task
    ⟨ row = nextRow; nextRow++; ⟩
    if (row >= n)
      break;
    compute inner products for c[row,*];
  }
}
```

Figure 3.20 Matrix multiplication using a bag of tasks.

we want to detect this, we could employ a shared counter, `done`, that is initially zero and that is incremented atomically before a worker executes `break`. Then if we want to have the last worker print the results, we could add the following code to the end of each worker:

```
if (done == n)
  print matrix c;
```

This uses `done` like a counter barrier.

3.6.2 Adaptive Quadrature

Recall that the quadrature problem is to approximate the integral of a function `f(x)` from `a` to `b`. With adaptive quadrature, one first computes the midpoint `m` between `a` and `b`. Then one approximates three areas: from `a` to `m`, from `m` to `b`, and from `a` to `b`. If the sum of the two smaller areas is within some acceptable tolerance of the larger area, then the approximation is considered good enough. If not, the larger problem is divided into two subproblems and the process is repeated.

We can use the bag-of-tasks paradigm as follows to implement adaptive quadrature. Here, a task is an interval to examine; it is defined by the end points of the interval, the value of the function at those endpoints, and the approximation of the area for that interval. Initially there is just one task, for the entire interval `a` to `b`.

Amazon
Exhibit 1011
Page 028

A worker repeatedly takes a task from the bag and executes it. In contrast to the matrix multiplication program, however, the bag may (temporarily) be empty and a worker might thus have to delay before getting a task. Moreover, executing one task will usually lead to producing two smaller tasks, which a worker needs to put back into the bag. Finally, it is more difficult to determine when all the work has been done because it is not sufficient simply to wait until the bag is empty. (Indeed, the bag will *be* empty the minute the first task is taken.) Rather, all the work is done only when both (1) the bag is empty and (2) each worker is waiting to get another task.

Figure 3.21 contains a program for adaptive quadrature using a bag of tasks. The bag is represented by a queue and a counter. Another counter keeps track of the number of idle workers. All the work has been finished when `size` is `0` and `idle` is `n`. Note that the program contains several atomic actions. These are needed to protect the critical sections that access the shared variables. All but one of the atomic actions are unconditional, so they can be protected using locks. However, the one `await` statement has to be implemented using the more elaborate protocol in Section 3.2 or using a more powerful synchronization mechanism such as semaphores or monitors.

The program in Figure 3.21 makes excessive use of the bag of tasks. In particular, when a worker decides to generate two tasks, it puts both of them in the bag, then loops around and takes a new task out of the bag (perhaps even one that it just put in). Instead, we could have a worker put one task in the bag and keep the other. Once the bag contains enough work to lead to a balanced computational load, a further optimization would be to have a worker execute a task fully, using sequential recursion, rather than putting more tasks into the bag.

Historical Notes

The critical section problem was first described by Edsger Dijkstra [1965]. Because the problem is fundamental, it has been studied by scores of people who have published literally hundreds of papers on the topic. This chapter has presented four of the most important solutions. (Raynal [1986] has written an entire book on mutual exclusion algorithms.) Although devising busy-waiting solutions was early on mostly an academic exercise—since busy waiting is inefficient on a single processor—the advent of multiprocessors has spurred renewed interest in such solutions. Indeed, all multiprocessors now provide instructions that support at least one busy-waiting solution. The exercises at the end of this chapter describe most of these instructions.

Dijkstra's paper [1965] presented the first `n`-process software solution. It is an extension of the first two-process solution designed by Dutch mathematician T. Dekker (see the exercises). However, Dijkstra's original formulation of the

Amazon
Exhibit 1011
Page 029

```
type task = (double left, right, fleft, fright, lrarea);
queue bag(task);        # the bag of tasks
int size;               # number of tasks in bag
int idle = 0;           # number of idle workers
double total = 0.0;     # the total area

compute approximate area from a to b;
insert task (a, b, f(a), f(b), area) in the bag;
count = 1;

process Worker[w = 1 to PR] {
  double left, right, fleft, fright, lrarea;
  double mid, fmid, larea, rarea;
  while (true) {
    # check for termination
    ⟨ idle++;
      if (idle == n && size == 0) break; ⟩
    # get a task from the bag
    ⟨ await (size > 0)
      remove a task from the bag;
      size--; idle--; ⟩
    mid = (left+right) / 2;
    fmid = f(mid);
    larea = (fleft+fmid) * (mid-left) / 2;
    rarea = (fmid+fright) * (right-mid) / 2;
    if (abs((larea+rarea) - lrarea) > EPSILON) {
      ⟨ put (left, mid, fleft, fmid, larea) in the bag;
        put (mid, right, fmid, fright, rarea) in the bag;
        size = size + 2; ⟩
    } else
      ⟨ total = total + lrarea; ⟩
  }
  if (w == 1)    # worker 1 prints the result
    printf("the total is %f\n", total);
}
```

Figure 3.21 Adaptive quadrature using a bag of tasks.

problem did not require the eventual entry property (3.5). Donald Knuth [1966] was the first to publish a solution that also ensures eventual entry.

The tie-breaker algorithm was discovered by Gary Peterson [1981]; it is often called Peterson's algorithm as a result. This algorithm is particularly simple for two processes, unlike the earlier solutions of Dekker, Dijkstra, and others.

Amazon
Exhibit 1011
Page 030

Peterson's algorithm also generalizes readily to an `n`-process solution, as shown in Figure 3.7. That solution requires that a process go through all `n-1` stages, even if no other process is trying to enter its critical section. Block and Woo [1990] present a variation that requires only `m` stages if just `m` processes are contending for entry (see the exercises).

The bakery algorithm was devised by Leslie Lamport [1974]. (Figure 3.11 contains an improved version that appeared in Lamport [1979].) In addition to being more intuitive than earlier critical section solutions, the bakery algorithm allows processes to enter in essentially FIFO order. It also has the interesting property that it is tolerant of some hardware failures. First, if one process reads `turn[i]` while another process is setting it in the entry protocol, the read can return any value between `1` and the value being written. Second, a process `CS[i]` may fail at any time, assuming it immediately sets `turn[i]` to `0`. However, the value of `turn[i]` can become unbounded in Lamport's solution if there is always at least one process in its critical section.

The ticket algorithm is the simplest of the `n`-process solutions to the critical section problem; it also allows processes to enter in the order in which they draw numbers. The ticket algorithm can in fact be viewed as an optimization of the bakery algorithm. However, it requires a Fetch-and-Add instruction, something that few machines provide (see Almasi and Gottlieb [1994] for machines that do).

Harry Jordan [1978] was one of the first to recognize the importance of barrier synchronization in parallel iterative algorithms; he is credited with coining the term barrier. Although barriers are not nearly as commonplace as critical sections, they too have been studied by dozens of people who have developed several different implementations. The combining tree barrier in Figures 3.13 and 3.14 is similar to one devised by Yew, Tzeng, and Lawrie [1987]. The butterfly barrier was devised by Brooks [1986]. Hensgen, Finkel, and Manber [1988] developed the dissemination barrier; their paper also describes a tournament barrier, which is similar in structure to a combining tree. Gupta [1989] describes a "fuzzy barrier," which includes a region of statements a process can execute while it waits for a barrier; his paper also describes a hardware implementation of fuzzy barriers.

As described in several places in the chapter, busy-waiting implementations of locks and barriers can lead to memory and interconnection-network contention. Thus it is important for delayed processes to spin on the contents of local memory, e.g., cached copies of variables. Chapter 8 of Hennessy and Patterson's architecture book [1996] discusses synchronization issues in multiprocessors, includes an interesting historical perspective, and provides an extensive set of references.

An excellent paper by Mellor-Crummy and Scott [1991] analyzes the performance of several lock and barrier protocols, including most of those described

Amazon
Exhibit 1011
Page 031

in the text. That paper also presents a new, list-based lock protocol and a scalable, tree-based barrier with only local spinning. A recent paper [McKenney 1996] describes different kinds of locking patterns in parallel programs and gives guidelines for selecting which locking primitives to use. Another recent paper [Savage et al. 1997] describes a tool called Eraser that can detect data races due to synchronization mistakes in lock-based multithreaded programs.

On multiprogrammed (time-sliced) systems, performance can suffer greatly if a process is preempted while holding a lock. One way to overcome this problem is to use *nonblocking*—or *lock free*—algorithms instead of using mutual exclusion locks. An implementation of a data structure is nonblocking if some process will complete an operation in a finite number of steps, regardless of the execution speed of other processes. Herlihy and Wing [1990] introduces this concept and presents a nonblocking implementation of concurrent queues. In Herlihy and Wing's implementation, the insert operation is also *wait free*—i.e., every insert operation will complete in a finite number of steps, again regardless of execution speeds. Herlihy [1990] presents a general methodology for implementing highly concurrent data structures, and Herlihy [1991] contains a comprehensive discussion of wait-free synchronization. A recent paper by Michael and Scott [1998] evaluates the performance of nonblocking implementations of several data structures; they also consider the alternative of lock-based implementations that are modified to be preemption safe. They conclude that nonblocking implementations are superior for those data structures for which they exist (e.g., queues), and that preemption-safe locks outperform normal locks on multiprogrammed systems.

Data parallel algorithms are most closely associated with massively parallel machines since those machines permit thousands of data elements to be operated on in parallel. Many of the earliest examples of data parallel algorithms were designed for the NYU Ultracomputer [Schwartz 1980]; all have the characteristic attribute that execution time is logarithmic in the size of the data. The Ultracomputer is a MIMD machine, and its designers realized the importance of efficient critical section and barrier synchronization protocols. Consequently, they implemented a Replace-and-Add operation in an early hardware prototype [Gottlieb et al. 1983]; this instruction adds a value to a memory word and then returns the result. However, more recent versions of the Ultracomputer provide a Fetch-and-Add operation instead [Almasi & Gottlieb 1994], which returns the value of a memory word before adding to it. This change was made because Fetch-and-Add, unlike Replace-and-Add, generalizes to any binary combining operator—for example, Fetch-and-Max or Fetch-and-And. This general kind of instruction is called a Fetch-and-Φ operation.

The introduction of the Connection Machine in the mid-1980s spurred renewed interest in data parallel algorithms. That machine was designed by

Amazon
Exhibit 1011
Page 032

Daniel Hillis [1985] as part of his doctoral dissertation at MIT. (Precious few dissertations have been as influential!) The Connection Machine was a SIMD machine, so barrier synchronization was automatically provided after every machine instruction. The Connection Machine also had thousands of processing elements; this enables a data parallel algorithm to have a very fine granularity—for example, a processor can be assigned to every element of an array. Hillis and Steele [1986] give an overview of the original Connection Machine and describe several interesting data parallel algorithms.

The bag-of-tasks paradigm for parallel computing was introduced by Carriero, Gelernter, and Leichter [1986]. That paper shows how to implement the task bag in the Linda programming notation, which we describe in Chapter 7. They called the associated programming model *replicated workers*, because any number of workers can share the bag. Some now call this the *work farm model*, because the tasks in the bag are farmed out to the workers. We prefer the phrase "bag of tasks" because it characterizes the essence of the approach: a shared bag of tasks.

References

Almasi, G. S., and A. Gottlieb. 1994. *Highly Parallel Computing*, 2nd ed. Menlo Park, CA: Benjamin/Cummings.

Block, K., and T.-K. Woo. 1990. A more efficient generalization of Peterson's mutual exclusion algorithm. *Information Processing Letters* 35 (August): 219–22.

Brooks, E. D., III. 1986. The butterfly barrier. *Int. Journal of Parallel Prog.* 15, 4 (August): 295–307.

Carriero, N., D. Gelernter, and J. Leichter. 1986. Distributed data structures in Linda. *Thirteenth ACM Symp. on Principles of Prog. Langs.*, January, pp. 236–42.

Dijkstra, E. W. 1965. Solution of a problem in concurrent programming control. *Comm. ACM* 8, 9 (September): 569.

Gottlieb, A., B. D. Lubachevsky, and L. Rudolph. 1983. Basic techniques for the efficient coordination of very large numbers of cooperating sequential processors. *ACM Trans. on Prog. Languages and Systems* 5, 2 (April): 164–89.

Gupta, R. 1989. The fuzzy barrier: a mechanism for high speed synchronization of processors. *Third Int. Conf. on Architectural Support for Prog. Languages and Operating Systems*, April, pp. 54–63.

Amazon
Exhibit 1011
Page 033

This statement evaluates `n` calls of `f` in parallel and assigns the results to shared array `a`. The corresponding C-Linda code is

```
for (i = 1; i <= n; i++)
  EVAL("a", i, f(i));
```

This forks `n` process tuples; each evaluates one call of `f(i)`. When a process tuple terminates, it turns into a data tuple containing the array name, an index, and the value of that element of `a`.

TS can also be used to implement conventional message passing. For example, a process can send a message to a channel `ch` by executing

```
OUT("ch", expressions);
```

Again, the name of a shared object is turned into a tag on a tuple. Another process can receive the message by executing

```
IN("ch", variables);
```

However, this statement extracts any matching tuple. If it is important that messages be received in the order they are sent, the programmer has to impose an ordering. This can be done by using counters to put sequence numbers on messages, and then reading them in sequence-number order.

7.7.2 Example: Prime Numbers with a Bag of Tasks

In Section 7.6, we showed how to generate prime numbers using the Sieve of Eratosthenes. In that approach, candidate numbers pass through a pipeline of filter processes. Each filter holds one prime factor and checks whether that prime divides each candidate; if not, the candidate might be prime so it is passed to the next filter process in the pipeline.

We can also generate primes using the bag-of-tasks paradigm (which was introduced in Section 3.6). As usual, the workers share a bag of tasks. In this case, the bag contains the candidate numbers; it is stored in tuple space. A manager process deposits candidates into the bag and gathers results.

Each worker repeatedly removes a candidate from the bag and determines whether it is prime by dividing it by smaller primes. But how can a worker acquire all the primes it needs? The answer is to check candidates in strictly increasing order. In this way, the prime factors needed to check a new candidate will already have been generated before they are needed. A worker might have to delay—waiting for another worker to generate a smaller prime—but workers will not deadlock since all smaller primes will eventually be generated.

Amazon
Exhibit 1011
Page 034

Figure 7.16 contains a C-Linda program that generates the first `limit` primes using a bag of tasks stored in tuple space. The structure of a C-Linda program is essentially that of a C program. However, the main routine is named `real_main` rather than `main`.

```
# include "linda.h"
# define LIMIT 1000      /* upper bound for limit */

void worker() {
  int primes[LIMIT] = {2,3};  /* table of primes */
  int numPrimes = 1, i, candidate, isprime;

  /* repeatedly get candidates and check them */
  while(true) {
    if (RDP("stop"))  /* check for termination */
      return;
    IN("candidate", ?candidate);     /* get candidate */
    OUT("candidate", candidate+2);  /* output next one */
    i = 0; isprime = 1;
    while (primes[i]*primes[i] <= candidate) {
      if (candidate%primes[i] == 0) { /* not prime */
        isprime = 0; break;
      }
      i++;
      if (i > numPrimes) {  /* need another prime */
        numPrimes++;
        RD("prime", numPrimes, ?primes[numPrimes]);
      }
    }
    /* tell manager the result */
    OUT("result", candidate, isprime);
  }
}

real_main(int argc, char *argv[]) {
  int primes[LIMIT] = {2,3};  /* my table of primes */
  int limit, numWorkers, i, isprime;
  int numPrimes = 2, value = 5;
  limit = atoi(argv[1]);  /* read command line */
  numWorkers = atoi(argv[2]);
```

Amazon
Exhibit 1011
Page 035

```
  /* create workers and put first candidate in bag */
  for (i = 1; i <= numWorkers; i++)
    EVAL("worker", worker());
  OUT("candidate", value);

  /* get results from workers in increasing order */
  while (numPrimes < limit) {
    IN("result", value, ?isprime);
    if (isprime) {   /* put value in table and TS */
      primes[numPrimes] = value;
      OUT("prime", numPrimes, value);
      numPrimes++;
    }
    value = value + 2;
  }
  /* tell workers to quit, then print the primes */
  OUT("stop");
  for (i = 0; i < limit; i++)
    printf("%d\n", primes[i]);
}
```

Figure 7.16 Prime number generation in C-Linda.

The main routine executes first as a single process. It begins by reading command-line arguments to determine how many primes to generate and how many workers to use. It then uses `EVAL` to fork the worker processes and uses `OUT` to deposit the first candidate, `5`, in TS.

The main routine then assumes the role of a manager process. In particular, it uses `IN` to receive the results from workers; these are received in order so that primes can be counted. When `real_main` receives a prime, it puts the prime and its count into TS and updates its local table. For example, `5` is the third prime, `7` is the fourth, and so on.

The program terminates when `real_main` has received `limit` primes. The manager tells the workers to stop by depositing a `stop` token in TS, then it prints the primes stored in its local table.

Each worker process repeatedly gets a candidate number from TS, checks its primality, and sends the result to the manager by depositing the result in TS. After a worker extracts a candidate number using `IN`, it deposits the next odd candidate into TS. Thus, there is at most one candidate number in TS at a time, and the candidates are increasing odd integers.

Amazon
Exhibit 1011
Page 036

Each worker keeps a local table of prime factors. It extends this table when it needs another prime factor to check a candidate number. It gets the prime by reading it from TS. To avoid deadlock, workers read primes in strictly increasing order.

On each iteration, a worker uses the conditional read primitive, `RDP`, to see if it should stop. This primitive returns true if there is a `stop` tuple in TS; otherwise it returns false. Since the workers will not see the `stop` tuple until after the manager has acquired `limit` primes, more candidates than necessary will be checked. This is hard to avoid without having the processes exchange many more messages.

The program in Figure 7.16 is not the most efficient one we could devise. This is because each candidate is deposited and extracted twice—once in a `candidate` tuple and then in a `result` tuple. Accessing TS is much less efficient than accessing local variables. To increase the use of local variables, we could have the manager deposit ranges of candidates—say 10 or 20 numbers at a time. We could then have the workers compute all primes in a given range and send the collective result to the manager. We could also seed the tables of primes in the manager and workers with more of the small primes.

7.8 Case Study: MPI

The Message Passing Interface (MPI) is a library of message-passing routines. When MPI is used, the processes in a distributed program are written in a sequential language such as C or Fortran; they communicate and synchronize by calling functions in the MPI library.

The MPI application programmer's interface (API) was defined in the mid-1990s by a large group of people from academia, government, and industry. The interface reflects people's experiences with earlier message-passing libraries, such as PVM. The goal of the group was to develop a single library that could be implemented efficiently on the variety of multiple processor machines. MPI has now become the de facto standard, and several implementations exist.

MPI programs have what is called an SPMD style—single program, multiple data. In particular, every processor executes a copy of the same program. (In an application written in `C`, the `main` routine in the program is started on every processor.) Each instance of the program can determine its own identity and hence take different actions. The instances interact by calling MPI library functions. The MPI functions support process-to-process communication, group communication, setting up and managing communication groups, and interacting with the environment.

Amazon
Exhibit 1011
Page 037

The first three paradigms are commonly used in parallel computations; the other four arise in distributed systems. We illustrate how the paradigms can be used to solve a variety of problems, including sparse matrix multiplication, image processing, distributed matrix multiplication, computing the topology of a network, distributed mutual exclusion, distributed termination detection, and decentralized dining philosophers. We later use the three parallel computing paradigms to solve the scientific computing problems in Chapter 11. Additional applications are described in the exercises, including sorting and the traveling salesman.

9.1 Manager/Workers (Distributed Bag of Tasks)

We introduced the bag-of-tasks paradigm in Section 3.6 and showed how to implement it using shared variables for communication and synchronization. Recall the basic idea: Several worker processes share a bag that contains independent tasks. Each worker repeatedly removes a task from the bag, executes it, and possibly generates one or more new tasks that it puts into the bag. The benefits of this approach to implementing a parallel computation are that it is easy to vary the number of workers and it is relatively easy to ensure that each does about the same amount of work.

Here we show how to implement the bag-of-tasks paradigm using message passing rather than shared variables. This is done by employing a manager process to implement the bag, hand out tasks, collect results, and detect termination. The workers get tasks and deposit results by communicating with the manager. Thus the manager is essentially a server process and the workers are clients.

The first example below shows how to multiply sparse matrices (matrices for which most entries are zero). The second example revisits the quadrature problem and uses a combination of static and adaptive integration intervals. In both examples, the total number of tasks is fixed, but the amount of work per task is variable.

9.1.1 Sparse Matrix Multiplication

Assume that `A` and `B` are $n \times n$ matrices. The goal, as before, is to compute the matrix product $A \times B = C$. This requires computing n^2 inner products. Each inner product is the sum (plus reduction) of the pairwise products of two vectors of length `n`.

A matrix is said to be *dense* if most entries are nonzero and it is *sparse* if most entries are zero. If `A` and `B` are dense matrices, then `C` will also be dense

Amazon
Exhibit 1011
Page 038

(unless there is considerable cancellation in the inner products). On the other hand, if either **A** or **B** is sparse, then **C** will also be sparse. This is because each zero in **A** or **B** will lead to a zero in **n** vector products and hence will not contribute to the result of **n** of the inner products. For example, if some row of **A** contains all zeros, then the corresponding row of **C** will also contain all zeros.

Sparse matrices arise in many contexts, such as numerical approximations to partial differential equations and large systems of linear equations. A tridiagonal matrix is an example of a sparse matrix; it has zeros everywhere except on the main diagonal and the two diagonals immediately above and below the main diagonal. When we know that matrices are sparse, we can save space by storing only the nonzero entries, and we can save time when multiplying matrices by ignoring entries that are zero.

We will represent sparse matrix **A** as follows by storing information about the rows of **A**:

```
int lengthA[n];
pair *elementsA[n];
```

The value of **lengthA[i]** is the number of nonzero entries in row **i** of **A**. Variable **elementsA[i]** points to a list of the nonzero entries in row **i**. Each entry is represented by a pair (record): an integer column index and a double precision data value. Thus if **lengthA[i]** is **3**, then there are three pairs in the list **elementsA[i]**. These will be ordered by increasing values of the column indexes. As a concrete example, consider the following:

```
lengthA  elementsA

  1      (3, 2.5)
  0
  0
  2      (1, -1.5)  (4, 0.6)
  0
  1      (0, 3.4)
```

This represents a six by six matrix containing four nonzero elements: one in row **0** column **3**, one in row **3** column **1**, one in row **3** column **4**, and one in row **5**, column **0**.

We will represent matrix **C** in the same way as **A**. However, to facilitate matrix multiplication, we will represent matrix **B** by columns rather than rows. In particular, **lengthB** indicates the number of nonzero elements in each column of **B**, and **elementsB** contains pairs of row index and data value.

Computing the matrix product of **A** and **B** requires as usual examining the n^2 pairs of rows and columns. With sparse matrices, the most natural size for a

Amazon
Exhibit 1011
Page 039

```
module Manager
  type pair = (int index, double value);
  op getTask(result int row, len; result pair [*]elems);
  op putResult(int row, len; pair [*]elems);
body Manager
  int lengthA[n], lengthC[n];
  pair *elementsA[n], *elementsC[n];
  # matrix A is assumed to be initialized
  int nextRow = 0, tasksDone = 0;
  process manager {
    while (nextRow < n or tasksDone < n) {
      # more tasks to do or more results needed
      in getTask(row, len, elems) ->
          row = nextRow;
          len = lengthA[i];
          copy pairs in *elementsA[i] to elems;
          nextRow++;
      [] putResult(row, len, elems) ->
          lengthC[row] = len;
          copy pairs in elems to *elementsC[row];
          tasksDone++;
      ni
    }
  }
end Manager
```

Figure 9.1 (a) Sparse matrix multiplication: Manager process.

task is one row of the result matrix `C`, because an entire row is represented by a single vector of (column, value) pairs. Thus there are as many tasks as there are rows of `A`. (An obvious optimization would be to skip over rows of `A` that are entirely zeros—namely, those for which `lengthA[i]` is `0`, because the corresponding rows of `C` would also be all zeros. This is not likely in realistic applications, however.)

Figure 9.1 contains code to implement sparse matrix multiplication using a manager and several worker processes. We assume that matrix `A` is already initialized in the manager and that each worker contains an initialized copy of matrix `B`. The processes interact using the rendezvous primitives of Chapter 8, because that leads to the simplest program. To use asynchronous message passing, the manager would be programmed in the style of an active monitor (Section 7.3) and the `call` in the workers would be replaced by `send` then `receive`.

Amazon
Exhibit 1011
Page 040

```
process worker[w = 1 to numWorkers] {
  int lengthB[n];
  pair *elementsB[n];  # assumed to be initialized
  int row, lengthA, lengthC;
  pair *elementsA, *elementsC;
  int r, c, na, nb;  # used in computing
  double sum;        #   inner products
  while (true) {
    # get a row of A, then compute a row of C
    call getTask(row, lengthA, elementsA);
    lengthC = 0;
    for [i = 0 to n-1]
      INNER_PRODUCT(i);  # see body of text
    send putResult(row, lengthC, elementsC);
  }
}
```

Figure 9.1 (b) Sparse matrix multiplication: Worker processes.

The manager process in Figure 9.1 (a) implements the bag and accumulates results. It services two operations: `getTask` and `putResult`. Integer `nextRow` identifies the next task—namely, the next row. Integer `tasksDone` counts the number of results that have been returned. When a task is handed out, `nextRow` is incremented; when a result is returned, `tasksDone` is incremented. When both values are equal to `n`, all tasks have been completed.

The code for the worker processes is shown in Figure 9.1 (b). Each worker repeatedly gets a new task, performs it, then sends the result to the manager. A task consists of computing a row of the result matrix `C`, so a worker executes a `for` loop to compute `n` inner products, one for every column of `B`. However, the code for computing an inner product of two sparse vectors differs significantly from the loop for computing an inner product of two dense vectors. In particular, the meaning of `INNER_PRODUCT(i)` in the worker code is

```
sum = 0.0; na = 1; nb = 1;
c = elementsA[na]->index;      # column in row of A
r = elementsB[i][nb]->index;   # row in column of B
while (na <= lengthA and nb <= lengthB) {
  if (r == c) {
    sum += elementsA[na]->value *
             elementsB[i][nb]->value;
    na++; nb++;
```

Amazon
Exhibit 1011
Page 041

```
      c = elementsA[na]->index;
      r = elementsB[i][nb]->index;
    } else if (r < c) {
      nb++; r = elementsB[i][nb]->index;
    } else { # r > c
      na++; c = elementsA[na]->index;
    }
  }
  if (sum != 0.0) {  # extend row of C
    elementsC[lengthC] = pair(i, sum);
    lengthC++;
  }
```

The basic idea is to scan through the sparse representations for the row of `A` and column `i` of `B`. (The above code assumes that the row and column each contain at least one nonzero element.) The inner product will be nonzero only if there are pairs of values that have the *same* column index `c` in `A` and row index `r` in `B`. The `while` loop above finds all these pairs and adds up their products. If the `sum` is nonzero when the loop terminates, then a new pair is added to the vector representing the row of `C`. (We assume that space for the elements has already been allocated.) After all `n` inner products have been computed, a worker sends the row of `C` to the manager, then calls the manager to get another task.

9.1.2 Adaptive Quadrature Revisited

The quadrature problem was introduced in Section 1.5, where we presented static (iterative) and dynamic (recursive) algorithms for approximating the integral of function `f(x)` from `a` to `b`. Section 3.6 showed how to implement adaptive quadrature using a shared bag of tasks.

Here we use a manager to implement a distributed bag of tasks. However, we use a combination of the static and dynamic algorithms rather than the "pure" adaptive quadrature algorithm used in Figure 3.21. In particular, we divide the interval from `a` to `b` into a fixed number of subintervals and then use the adaptive quadrature algorithm within each subinterval. This combines the simplicity of an iterative algorithm with the greater accuracy of an adaptive algorithm. In particular, by using a static number of tasks, the manager and workers are simpler and there are far fewer interactions between them.

Figure 9.2 contains the code for the manager and workers. Because the manager is essentially a server process, we again use rendezvous for interaction

Amazon
Exhibit 1011
Page 042

```
module Manager
  op getTask(result double left, right);
  op putResult(double area);
body Manager
  process manager {
    double a, b;          # interval to integrate
    int numIntervals;     # number of intervals to use
    double width = (b-a)/numIntervals;
    double x = a, totalArea = 0.0;
    int tasksDone = 0;
    while (tasksDone < numIntervals) {
      in getTask(left, right) st x < b ->
          left = x; x += width; right = x;
      [] putResult(area) ->
          totalArea += area;
          tasksDone++;
      ni
    }
  print the result totalArea;
  }
end Manager

double f() { ... }          # function to integrate
double quad(...) { ... }    # adaptive quad function

process worker[w = 1 to numWorkers] {
  double left, right, area = 0.0;
  double fleft, fright, lrarea;
  while (true) {
    call getTask(left, right);
    fleft = f(left); fright = f(right);
    lrarea = (fleft + fright) * (right - left) / 2;
    # calculate area recursively as shown in Section 1.5
    area = quad(left, right, fleft, fright, lrarea);
    send putResult(area);
  }
}
```

Figure 9.2 Adaptive quadrature using manager/workers paradigm.

Amazon
Exhibit 1011
Page 043

between the manager and workers. Thus the manager has the same structure as that shown in Figure 9.1 (a) and again exports two operations, `getTask` and `putResult`. However, the operations have different parameters, because now a task is defined by the endpoints of an interval, `left` and `right`, and a result consists of the area under `f(x)` over that interval. We assume the values of `a`, `b`, and `numIntervals` are given, perhaps as command-line arguments. From these values, the manager computes the width of each interval. The manager then loops, accepting calls of `getTask` and `putResult`, until it has received one result for each interval (and hence task). Notice the use of a synchronization expression "`st x< b`" in the arm for `getTask` in the input statement; this prevents `getTask` from handing out another task when the bag is empty.

The worker processes in Figure 9.2 share or have their own copies of the code for functions `f` and `quad`, where `quad` is the recursive function given in Section 1.5. Each worker repeatedly gets a task from the manager, calculates the arguments needed by `quad`, calls `quad` to approximate the area under `f(left)` to `f(right)`, then sends the result to the manager.

When the program in Figure 9.2 terminates, every worker will be blocked at its call of `getTask`. This is often harmless, as here, but at times it can indicate deadlock. We leave to the reader modifying the program so that the workers terminate normally. (*Hint:* Make `getTask` a function that returns `true` or `false`.)

In this program the amount of work per task varies, depending on how rapidly the function `f` varies. Thus, if there are only about as many tasks as workers, the computational load will almost certainly be unbalanced. On the other hand, if there are too many tasks, then there will be unnecessary interactions between the manager and workers, and hence unnecessary overhead. The ideal is to have just enough tasks so that, on average, each worker will be responsible for about the same total amount of work. A reasonable heuristic is to have from two to three times as many tasks as workers, which here means that `numIntervals` should be two to three times larger than `numWorkers`.

9.2 Heartbeat Algorithms

The bag-of-tasks paradigm is useful for solving problems that have a fixed number of independent tasks or that result from use of the divide-and-conquer paradigm. The *heartbeat paradigm* is useful for many data parallel iterative applications. In particular, it can be used when the data is divided among workers, each is responsible for updating a particular part, and new data values depend on values held by workers or their immediate neighbors. Applications include

Amazon
Exhibit 1011
Page 044

```
process Worker[w = 1 to PR] {
  # run the simulation with time steps of DT
  for [time = start to finish by DT] {
    calculateForces(w);
    barrier(w);
    moveBodies(w);
    barrier(w);
  }
}
```

Figure 11.11 Shared-variable program for the n-body problem.

so that the workload is balanced. We also need to minimize communication overhead or at least to make it small relative to the amount of useful work. These are challenging issues because the n-body problem involves calculating forces between every pair of bodies, and hence every worker process needs to interact with all the others.

A useful starting point is to consider whether we can use any of the three process interaction paradigms at the start of Chapter 9: manager/workers (distributed bag of tasks), heartbeat, and pipeline. There are well-defined tasks and load balancing is an issue, so the manager/workers paradigm is appropriate. We can also use a heartbeat algorithm by dividing up the bodies and having workers exchange information; however, it is not sufficient to have each worker just exchange information with one or two neighbors. Finally, we can also use a pipeline algorithm and have bodies flow from worker to worker. Below we describe how to solve the n-body problem using each of these paradigms, then we discuss the tradeoffs between them.

Manager/Workers Program

First consider using the manager/workers paradigm (Section 9.1). With this approach the manager maintains a bag of tasks; workers repeatedly get a task, do it, then return the result to the manager. For the n-body problem, there are two phases. In the first, the tasks are to calculate the forces between every pair of bodies. In the second, the tasks are to move the bodies. The majority of the computation—and the part that can be imbalanced—is calculating forces. Hence it makes sense to use dynamic tasks for the force calculations but to use static tasks (one per worker) for moving bodies.

Amazon
Exhibit 1011
Page 045

Suppose there are `PR` processors and that we want to use `PR` worker processes. (Hence, the manager will execute on the same processor as one of the workers.) Assume for simplicity that `n` is a multiple of `PR`. When using the manager/workers paradigm, we want to have at least twice as many tasks as workers to balance the load, but we do not want too many tasks or the workers will spend too much time communicating with the manager. One way to divide up the force calculations for the `n` bodies is as follows:

> Divide the bodies into `PR` blocks of size `n/PR`. The first block contains the first `n/PR` bodies, the second the next `n/PR` bodies, and so on.
>
> Form pairs (i, j) for each different combination of block numbers. There will be `PR*(PR+1)/2` such pairs—the sum of the integers from `1` to `PR`.
>
> Let each pair represent a task—for example, the task for pair (i, j) is to calculate the forces between the bodies in block i and those in block j.

As a concrete example, suppose `PR` is four. Then the ten tasks are the pairs

$$(1, 1)\ (1, 2)\ (1, 3)\ (1, 4)\ (2, 2)\ (2, 3)\ (2, 4)\ (3, 3)\ (3, 4)\ (4, 4)$$

The manager process initializes the bag with the pairs defining the tasks. Workers repeatedly get a task from the manager and calculate the forces specified by the task. To do so, a worker needs to know the positions, velocities, and masses of the bodies in one or two blocks. The manager could send these along with the task, but we can greatly reduce the length of messages by having each worker keep its own copy of the data for each body. Similarly, we can get rid of the need to send results to the manager if each worker keeps track of the forces it has calculated for each body.

After calculating all forces—namely when the bag of tasks is empty—the workers need to exchange forces and then move the bodies. The simplest static assignment of work for the move-bodies phase is to have each worker move the bodies in one block: worker 1 moves block 1, worker 2 moves block 2, and so on. Hence, each worker `w` needs the force vectors for the bodies in block `w`. We could use the manager to collect and disseminate the forces calculated by each worker, but it will be more efficient if each worker sends the forces for a block of bodies directly to the worker that will move those bodies. Alternatively, if we have global communication primitives—such as those in the MPI library—the workers can use them to broadcast and reduce (add) the values in the force vectors. After each worker moves its block of bodies, the workers need to exchange the new positions and velocities of the bodies. Again we could use point-to-point messages or global communication to effect this exchange.

Amazon
Exhibit 1011
Page 046

Figure 11.12 contains an outline of the code for a manager/workers program. The outer loop in each process executes once for each time step in the simulation. The inner loop in the manager process iterates `numTasks+PR` times, where `numTasks` is the number of tasks in the bag. On the last `PR` iterations, the bag will be empty, so the manager sends the pair (0, 0) as a "bag is empty" sentinel. When each of the `PR` workers gets this sentinel, it breaks out of the loop that calculates forces.

Heartbeat Program

A heartbeat algorithm (Section 9.2) is the message-passing analog of a shared-variable algorithm that has computational phases followed by barriers. The shared-variable program in Figure 11.11 has this structure, so we can modify it as follows to use message passing: (1) each worker is assigned a subset of the bodies; (2) every worker first calculates the forces for its set of bodies then exchanges forces with the other workers; (3) every worker moves its bodies; (4) the workers exchange the new positions and velocities of the bodies. This pattern is repeated for each time step in the simulation.

We can assign bodies to worker processes using any of the allocation patterns described in Figure 11.10: blocks, stripes, or reverse stripes. If we use a stripes or reverse stripes allocation, then the computational load will be much more balanced than if we use fixed-size blocks. However, each worker would need to have its own copy of the position, velocity, force, and mass vectors. Moreover, each worker would need to exchange entire vectors with every other worker after each phase. This leads to a large number of long messages.

If we assign blocks of bodies to workers, we will use about half as many messages and they will be shorter. (We will also need less local storage in the workers.) To see why, consider the example of four workers and ten tasks described in the previous section. With a block assignment, worker 1 handles the first four tasks: (1, 1), (1, 2), (1, 3), and (1, 4). Namely, it handles all force calculations involving the bodies in block 1. To do so, it needs positions and velocities for the bodies in blocks 2, 3, and 4, and it needs to return forces for those blocks. However, worker 1 never needs to tell another worker the position or velocity of its bodies and it never needs to get force information from any other worker. At the other extreme, worker 4 is responsible only for the task (4, 4). It does not need to know the position or velocity of any other body and it does not need to send forces to any other worker.

If the blocks are all the same size, then the workload will be unbalanced, which would more than offset any gain from sending fewer messages. However,

Amazon
Exhibit 1011
Page 047

Sorting algorithms (cont.)
merge network, 300–1
quicksort, 17
Soundness, of a logic, 58
Spanning tree, 445
Sparse matrix multiplication, 424–28
Spatial locality, 5
Speedup, of a parallel computation, 528
Spin lock, 99
Spin loop, 55
Spinning process, 57
Split binary semaphore, 159
SPMD (single program multiple data), 316, 340, 541
SR language, 406–11, 615–16
history and references, 414
SST disk scheduling, 227, 310
Stable marriage problem, 90, 356, 420
Stable prefix, of a message queue, 455
StarMod language, 412
Starvation, absence of. *See* Fairness
State, of a
process, 267
program, 42
Strength reduction transformation, 536
Strip mining, of loops, 611
Stripes, allocation by, 557
Strips, allocation by, 557
Strongly fair scheduling policy, 76
Structure of distributed programs. *See* Distributed program structures
Successive over relaxation, 546–49
Symmetric barrier, 121
Symmetric multiprocessor (SMP), 6
Synchronization, 2. *See also* Condition synchronization; Mutual exclusion
specified using `await`, 54–55
used for avoiding interference, 69–70
Synchronizing Resources. *See* SR language
Synchronous execution, 131
Synchronous message passing, 318–20. *See also* CSP language
implementation of, 496–503
Synchronous multiprocessor, 131
Synchronous `send` statement, 318

Task parallel program, 11
Temporal locality, 5
Temporal logic, 80
Termination detection problem, 460–65
Test-and-test-and-set protocol, 101
Test-and-Set instruction, 99
Textual substitution, 60
Thread of control, 2, 184
Ticket algorithm, 108–11, 137
Tie-breaker algorithm, 104–8, 136
Tightly coupled machine, 9
Tiling, of loops, 612
Time server. *See also* Interval timer
using remote procedure call, 365–67
using rendezvous, 378
Timestamp, 453
Token-passing algorithm, 423, 457–65
decentralized dining philosophers, 467–71
distributed mutual exclusion, 457–60
termination detection, 460–65
Topology of a network problem, 448
Total correctness, 43, 59
Tournament barrier, 146
Transformations, in parallelizing compilers, 608
Transputer machine, 329
Traveling salesman problem, 479–80
Tree code, 570
Tree-structured barrier, 118–20
Triangular matrix, 574
Triple, in a programming logic, 59
Tuple space, 334. *See also* Linda language
Turing Plus language, 251

UMA multiprocessor, 6
Unconditional atomic action, 55
Unconditionally fair scheduling policy, 75
Uniform memory access time, 6
Unisex bathroom problem, 200
Unix pipes, 20
Unroll and jam transformation, 610

V cycle, 552
`V` operation, 154–55. *See also* Semaphore
meaning of name, 188
Visualization tools, 634

W cycle, 552
Wait-free synchronization, 138
`wait` statement, 207–9
implementation of, 281, 285
Water molecule problem, 198, 257
Wave front synchronization, 580, 612
Weakened assertions, 66–68
Weakly fair scheduling policy, 75
Weighted voting, 392–93, 413
Work farm model, 139. *See also* Bag-of-tasks paradigm; Manager/workers paradigm
Writers' preference, 177
Write set, 45, 65

ZPL language, 615, 621–23, 640

Amazon
Exhibit 1011
Page 048

Parallel Programming/Concurrent Programming/Operating Systems

Foundations of Multithreaded, Parallel, and Distributed Programming

Gregory R. Andrews, The University of Arizona

Greg Andrews has produced a first-rate textbook for students that also serves as an authoritative and highly readable reference for professional software developers. I recommend it to anyone interested in stepping past the particular constructs found, for example, in the Java or Ada programming languages or the POSIX threads library, to gain a deeper understanding of underlying programming principles and practices. Andrews succeeds in capturing the conceptual unity of concurrent, distributed, and parallel programming, while still conveying the diversity of expression, specialized algorithms, and design and analysis techniques associated with each.

—**Professor Doug Lea**, State University of New York at Oswego, and author of *Concurrent Programming in Java: Design Principles and Patterns, Second Edition*

Andrews does a very good job of presenting the different methods of implementing concurrent systems at both low- and high-levels.

—**Professor Dyke Stiles**
Utah State University

The clarity of explanations and the use of the same problem in different paradigms to compare and contrast implementations [illegible] ples and accompanying explanations [illegible]

—**Professor Janet H**[illegible]
Illinois State Unive[illegible]

51249 Foundations of Multithre USED
02
FOUNDATIONS OF MULTITHREA
DED PARALLEL
PB
S05
114
524
-35752-6
99990
USED
0706357 4
3381860U
22 B

ADDISON-WESLEY

Addison-Wesley is an imprint of Addison Wesley Longman, Inc.