PARALLEL PROGRAMMING

Techniques and Applications Using Networked Workstations and Parallel Computers

Second Edition

BARRY WILKINSON MICHAEL ALLEN

PARALLEL PROGRAMMING

TECHNIQUES AND APPLICATIONS USING NETWORKED WORKSTATIONS AND PARALLEL COMPUTERS

2nd Edition

BARRY WILKINSON
University of North Carolina at Charlotte
Western Carolina University

MICHAEL ALLEN
University of North Carolina at Charlotte

PEARSON
Prentice Hall

Upper Saddle River, NJ 07458

Library of Congress CataIoging-in-Publication Data

CIP DATA AVAILABLE.

Vice President and Editorial Director, ECS: *Marcia Horton*
Executive Editor: *Kate Hargett*
Vice President and Director of Production and Manufacturing, ESM: *David W. Riccardi*
Executive Managing Editor: *Vince O'Brien*
Managing Editor: *Camille Trentacoste*
Production Editor: *John Keegan*
Director of Creative Services: *Paul Belfanti*
Art Director: *Jayne Conte*
Cover Designer: *Kiwi Design*
Managing Editor, AV Management and Production: *Patricia Burns*
Art Editor: *Gregory Dulles*
Manufacturing Manager: *Trudy Pisciotti*
Manufacturing Buyer: *Lisa McDowell*
Marketing Manager: *Pamela Hersperger*

© 2005, 1999 Pearson Education, Inc.
Pearson Prentice Hall
Pearson Education, Inc.
Upper Saddle River, NJ 07458

All rights reserved. No part of this book may be reproduced in any form or by any means, without permission in writing from the publisher.

Pearson Prentice Hall® is a trademark of Pearson Education, Inc.

The author and publisher of this book have used their best efforts in preparing this book. These efforts include the development, research, and testing of the theories and programs to determine their effectiveness. The author and publisher make no warranty of any kind, expressed or implied, with regard to these programs or the documentation contained in this book. The author and publisher shall not be liable in any event for incidental or consequential damages in connection with, or arising out of, the furnishing, performance, or use of these programs.

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

ISBN: 0-13-140563-2

Pearson Education Ltd., *London*
Pearson Education Australia Pty. Ltd., *Sydney*
Pearson Education Singapore, Pte. Ltd.
Pearson Education North Asia Ltd., *Hong Kong*
Pearson Education Canada, Inc., *Toronto*
Pearson Educación de Mexico, S.A. de C.V.
Pearson Education—Japan, *Tokyo*
Pearson Education Malaysia, Pte. Ltd.
Pearson Education, Inc., *Upper Saddle River, New Jersey*

To my wife, Wendy,
and my daughter, Johanna

Barry Wilkinson

To my wife, Bonnie

Michael Allen

Preface

The purpose of this text is to introduce parallel programming techniques. Parallel programming is programming multiple computers, or computers with multiple internal processors, to solve a problem at a greater computational speed than is possible with a single computer. It also offers the opportunity to tackle larger problems, that is, problems with more computational steps or larger memory requirements, the latter because multiple computers and multiprocessor systems often have more total memory than a single computer. In this text, we concentrate upon the use of multiple computers that communicate with one another by sending messages; hence the term *message-passing* parallel programming. The computers we use can be different types (PC, SUN, SGI, etc.) but must be interconnected, and a software environment must be present for message passing between computers. Suitable computers (either already in a network or capable of being interconnected) are very widely available as the basic computing platform for students, so that it is usually not necessary to acquire a specially designed multiprocessor system. Several software tools are available for message-passing parallel programming, notably several implementations of MPI, which are all freely available. Such software can also be used on specially designed multiprocessor systems should these systems be available for use. So far as practicable, we discuss techniques and applications in a system-independent fashion.

Second Edition. Since the publication of the first edition of this book, the use of interconnected computers as a high-performance computing platform has become widespread. The term "cluster computing" has come to be used to describe this type of computing. Often the computers used in a cluster are "commodity" computers, that is, low-cost personal computers as used in the home and office. Although the focus of this text, using multiple computers and processors for high-performance computing, has not been changed, we have revised our introductory chapter, Chapter 1, to take into account the move towards

commodity clusters and away from specially designed, self-contained, multiprocessors. In the first edition, we described both PVM and MPI and provided an appendix for each. However, only one would normally be used in the classroom. In the second edition, we have deleted specific details of PVM from the text because MPI is now a widely adopted standard and provides for much more powerful mechanisms. PVM can still be used if one wishes, and we still provide support for it on our home page.

Message-passing programming has some disadvantages, notably the need for the programmer to specify explicitly where and when the message passing should occur in the program and what to send. Data has to be sent to those computers that require the data through relatively slow messages. Some have compared this type of programming to assembly language programming, that is, programming using the internal language of the computer, a very low-level and tedious way of programming which is not done except under very specific circumstances. An alternative programming model is the shared memory model. In the first edition, shared memory programming was covered for computers with multiple internal processors and a common shared memory. Such shared memory multiprocessors have now become cost-effective and common, especially dual- and quad-processor systems. Thread programming was described using Pthreads. Shared memory programming remains in the second edition and with significant new material added including performance aspects of shared memory programming and a section on OpenMP, a thread-based standard for shared memory programming at a higher level than Pthreads. Any broad-ranging course on practical parallel programming would include shared memory programming, and having some experience with OpenMP is very desirable. A new appendix is added on OpenMP. OpenMP compilers are available at low cost to educational institutions.

With the focus of using clusters, a major new chapter has been added on shared memory programming on clusters. The shared memory model can be employed on a cluster with appropriate distributed shared memory (DSM) software. Distributed shared memory programming attempts to obtain the advantages of the scalability of clusters and the elegance of shared memory. Software is freely available to provide the DSM environment, and we shall also show that students can write their own DSM systems (we have had several done so). We should point out that there are performance issues with DSM. The performance of software DSM cannot be expected to be as good as true shared memory programming on a shared memory multiprocessor. But a large, scalable shared memory multiprocessor is much more expensive than a commodity cluster.

Other changes made for the second edition are related to programming on clusters. New material is added in Chapter 6 on partially synchronous computations, which are particularly important in clusters where synchronization is expensive in time and should be avoided. We have revised and added to Chapter 10 on sorting to include other sorting algorithms for clusters. We have added to the analysis of the algorithms in the first part of the book to include the computation/communication ratio because this is important to message-passing computing. Extra problems have been added. The appendix on parallel computational models has been removed to maintain a reasonable page count.

The first edition of the text was described as course text primarily for an undergraduate-level parallel programming course. However, we found that some institutions also used the text as a graduate-level course textbook. We have also used the material for both senior undergraduate-level and graduate-level courses, and it is suitable for beginning

graduate-level courses. For a graduate-level course, more advanced materials, for example, DSM implementation and fast Fourier transforms, would be covered and more demanding programming projects chosen.

Structure of Materials. As with the first edition, the text is divided into two parts. Part I now consists of Chapters 1 to 9, and Part II now consists of Chapters 10 to 13. In Part I, the basic techniques of parallel programming are developed. In Chapter 1, the concept of parallel computers is now described with more emphasis on clusters. Chapter 2 describes message-passing routines in general and particular software (MPI). Evaluating the performance of message-passing programs, both theoretically and in practice, is discussed. Chapter 3 describes the ideal problem for making parallel the embarrassingly parallel computation where the problem can be divided into independent parts. In fact, important applications can be parallelized in this fashion. Chapters 4, 5, 6, and 7 describe various programming strategies (partitioning and divide and conquer, pipelining, synchronous computations, asynchronous computations, and load balancing). These chapters of Part I cover all the essential aspects of parallel programming with the emphasis on message-passing and using simple problems to demonstrate techniques. The techniques themselves, however, can be applied to a wide range of problems. Sample code is usually given first as sequential code and then as parallel pseudocode. Often, the underlying algorithm is already parallel in nature and the sequential version has "unnaturally" serialized it using loops. Of course, some algorithms have to be reformulated for efficient parallel solution, and this reformulation may not be immediately apparent. Chapter 8 describes shared memory programming and includes Pthreads, an IEEE standard system that is widely available, and OpenMP. There is also a significant new section on timing and performance issues. The new chapter on distributed shared memory programming has been placed after the shared memory chapter to complete Part I, and the subsequent chapters have been renumbered.

Many parallel computing problems have specially developed algorithms, and in Part II problem-specific algorithms are studied in both non-numeric and numeric domains. For Part II, some mathematical concepts are needed, such as matrices. Topics covered in Part II include sorting (Chapter 10), numerical algorithms, matrix multiplication, linear equations, partial differential equations (Chapter 11), image processing (Chapter 12), and searching and optimization (Chapter 13). Image processing is particularly suitable for parallelization and is included as an interesting application with significant potential for projects. The fast Fourier transform is discussed in the context of image processing. This important transform is also used in many other areas, including signal processing and voice recognition.

A large selection of "real-life" problems drawn from practical situations is presented at the end of each chapter. These problems require no specialized mathematical knowledge and are a unique aspect of this text. They develop skills in the use of parallel programming techniques rather than simply teaching how to solve specific problems, such as sorting numbers or multiplying matrices.

Prerequisites. The prerequisite for studying Part I is a knowledge of sequential programming, as may be learned from using the C language. The parallel pseudocode in the text uses C-like assignment statements and control flow statements. However, students with only a knowledge of Java will have no difficulty in understanding the pseudocode,

because syntax of the statements is similar to that of Java. Part I can be studied immediately after basic sequential programming has been mastered. Many assignments here can be attempted without specialized mathematical knowledge. If MPI is used for the assignments, programs are usually written in C or C++ calling MPI message-passing library routines. The descriptions of the specific library calls needed are given in Appendix A. It is possible to use Java, although students with only a knowledge of Java should not have any difficulty in writing their assignments in C/C++.

In Part II, the sorting chapter assumes that the student has covered sequential sorting in a data structure or sequential programming course. The numerical algorithms chapter requires the mathematical background that would be expected of senior computer science or engineering undergraduates.

Course Structure. The instructor has some flexibility in the presentation of the materials. Not everything need be covered. In fact, it is usually not possible to cover the whole book in a single semester. A selection of topics from Part I would be suitable as an addition to a normal sequential programming class. We have introduced our first-year students to parallel programming in this way. In that context, the text is a supplement to a sequential programming course text. All of Part I and selected parts of Part II together are suitable as a more advanced undergraduate or beginning graduate-level parallel programming/computing course, and we use the text in that manner.

Home Page. A Web site has been developed for this book as an aid to students and instructors. It can be found at www.cs.uncc.edu/par_prog. Included at this site are extensive Web pages to help students learn how to compile and run parallel programs. Sample programs are provided for a simple initial assignment to check the software environment. The Web site has been completely redesigned during the preparation of the second edition to include step-by-step instructions for students using navigation buttons. Details of DSM programming are also provided. The new Instructor's Manual is available to instructors, and gives MPI solutions. The original solutions manual gave PVM solutions and is still available. The solutions manuals are available electronically from the authors. A very extensive set of slides is available from the home page.

Acknowledgments. The first edition of this text was the direct outcome of a National Science Foundation grant awarded to the authors at the University of North Carolina at Charlotte to introduce parallel programming in the first college year.[1] Without the support of the late Dr. M. Mulder, program director at the National Science Foundation, we would not have been able to pursue the ideas presented in the text. A number of graduate students worked on the original project. Mr. Uday Kamath produced the original solutions manual.

We should like to record our thanks to James Robinson, the departmental system administrator who established our local workstation cluster, without which we would not have been able to conduct the work. We should also like to thank the many students at UNC Charlotte who took our classes and helped us refine the material over many years. This

[1]National Science Foundation grant "Introducing parallel programming techniques into the freshman curricula," ref. DUE 9554975.

included "teleclasses" in which the materials for the first edition were classroom tested in a unique setting. The teleclasses were broadcast to several North Carolina universities, including UNC Asheville, UNC Greensboro, UNC Wilmington, and North Carolina State University, in addition to UNC Charlotte. Professor Mladen Vouk of North Carolina State University, apart from presenting an expert guest lecture for us, set up an impressive Web page that included "real audio" of our lectures and "automatically turning" slides. (These lectures can be viewed from a link from our home page.) Professor John Board of Duke University and Professor Jan Prins of UNC Chapel Hill also kindly made guest-expert presentations to classes. A parallel programming course based upon the material in this text was also given at the Universidad Nacional de San Luis in Argentina by kind invitation of Professor Raul Gallard.

The National Science Foundation has continued to support our work on cluster computing, and this helped us develop the second edition. A National Science Foundation grant was awarded to us to develop distributed shared memory tools and educational materials.[2] Chapter 9, on distributed shared memory programming, describes the work. Subsequently, the National Science Foundation awarded us a grant to conduct a three-day workshop at UNC Charlotte in July 2001 on teaching cluster computing,[3] which enabled us to further refine our materials for this book. We wish to record our appreciation to Dr. Andrew Bernat, program director at the National Science Foundation, for his continuing support. He suggested the cluster computing workshop at Charlotte. This workshop was attended by 18 faculty from around the United States. It led to another three-day workshop on teaching cluster computing at Gujarat University, Ahmedabad, India, in December 2001, this time by invitation of the IEEE Task Force on Cluster Computing (TFCC), in association with the IEEE Computer Society, India. The workshop was attended by about 40 faculty. We are also deeply in the debt to several people involved in the workshop, and especially to Mr. Rajkumar Buyya, chairman of the IEEE Computer Society Task Force on Cluster Computing who suggested it. We are also very grateful to Prentice Hall for providing copies of our textbook to free of charge to everyone who attended the workshops.

We have continued to test the materials with student audiences at UNC Charlotte and elsewhere (including the University of Massachusetts, Boston, while on leave of absence). A number of UNC-Charlotte students worked with us on projects during the development of the second edition. The new Web page for this edition was developed by Omar Lahbabi and further refined by Sari Ansari, both undergraduate students. The solutions manual in MPI was done by Thad Drum and Gabriel Medin, also undergraduate students at UNC-Charlotte.

We would like to express our continuing appreciation to Petra Recter, senior acquisitions editor at Prentice Hall, who supported us throughout the development of the second edition. Reviewers provided us with very helpful advice, especially one anonymous reviewer whose strong views made us revisit many aspects of this book, thereby definitely improving the material.

Finally, we wish to thank the many people who contacted us about the first edition, providing us with corrections and suggestions. We maintained an on-line errata list which was useful as the book went through reprints. All the corrections from the first edition have

[2]National Science Foundation grant "Parallel Programming on Workstation Clusters," ref. DUE 995030.

[3]National Science Foundation grant supplement for a cluster computing workshop, ref. DUE 0119508.

been incorporated into the second edition. An on-line errata list will be maintained again for the second edition with a link from the home page. We always appreciate being contacted with comments or corrections. Please send comments and corrections to us at wilkinson@email.wcu.edu (Barry Wilkinson) or cma@uncc.edu (Michael Allen).

BARRY WILKINSON
Western Carolina University

MICHAEL ALLEN
University of North Carolina, Charlotte

About the Authors

Barry Wilkinson is a full professor in the Department of Computer Science at the University of North Carolina at Charlotte, and also holds a faculty position at Western Carolina University. He previously held faculty positions at Brighton Polytechnic, England (1984–87), the State University of New York, College at New Paltz (1983–84), University College, Cardiff, Wales (1976–83), and the University of Aston, England (1973–76). From 1969 to 1970, he worked on process control computer systems at Ferranti Ltd. He is the author of *Computer Peripherals* (with D. Horrocks, Hodder and Stoughton, 1980, 2nd ed. 1987), *Digital System Design* (Prentice Hall, 1987, 2nd ed. 1992), *Computer Architecture Design and Performance* (Prentice Hall 1991, 2nd ed. 1996), and *The Essence of Digital Design* (Prentice Hall, 1997). In addition to these books, he has published many papers in major computer journals. He received a B.S. degree in electrical engineering (with first-class honors) from the University of Salford in 1969, and M.S. and Ph.D. degrees from the University of Manchester (Department of Computer Science), England, in 1971 and 1974, respectively. He has been a senior member of the IEEE since 1983 and received an IEEE Computer Society Certificate of Appreciation in 2001 for his work on the IEEE Task Force on Cluster Computing (TFCC) education program.

Michael Allen is a full professor in the Department of Computer Science at the University of North Carolina at Charlotte. He previously held faculty positions as an associate and full professor in the Electrical Engineering Department at the University of North Carolina at Charlotte (1974–85), and as an instructor and an assistant professor in the Electrical Engineering Department at the State University of New York at Buffalo (1968–74). From 1985 to 1987, he was on leave from the University of North Carolina at Charlotte while serving as the president and chairman of DataSpan, Inc. Additional industry experience includes electronics design and software systems development for Eastman Kodak, Sylvania Electronics, Bell of Pennsylvania, Wachovia Bank, and numerous other firms. He received B.S. and M.S. degrees in Electrical Engineering from Carnegie Mellon University in 1964 and 1965, respectively, and a Ph.D. from the State University of New York at Buffalo in 1968.

Contents

Preface ***v***

About the Authors ***xi***

PART I BASIC TECHNIQUES ***1***

CHAPTER 1 PARALLEL COMPUTERS ***3***

1.1 The Demand for Computational Speed 3

1.2 Potential for Increased Computational Speed 6
Speedup Factor 6
What Is the Maximum Speedup? 8
Message-Passing Computations 13

1.3 Types of Parallel Computers 13
Shared Memory Multiprocessor System 14
Message-Passing Multicomputer 16
Distributed Shared Memory 24
MIMD and SIMD Classifications 25

1.4 Cluster Computing 26
Interconnected Computers as a Computing Platform 26
Cluster Configurations 32
Setting Up a Dedicated "Beowulf Style" Cluster 36

1.5 Summary 38

Further Reading 38

Bibliography 39

Problems 41

CHAPTER 2 MESSAGE-PASSING COMPUTING 42

2.1 Basics of Message-Passing Programming 42
Programming Options 42
Process Creation 43
Message-Passing Routines 46

2.2 Using a Cluster of Computers 51
Software Tools 51
MPI 52
Pseudocode Constructs 60

2.3 Evaluating Parallel Programs 62
Equations for Parallel Execution Time 62
Time Complexity 65
Comments on Asymptotic Analysis 68
Communication Time of Broadcast/Gather 69

2.4 Debugging and Evaluating Parallel Programs Empirically 70
Low-Level Debugging 70
Visualization Tools 71
Debugging Strategies 72
Evaluating Programs 72
Comments on Optimizing Parallel Code 74

2.5 Summary 75

Further Reading 75

Bibliography 76

Problems 77

CHAPTER 3 EMBARRASSINGLY PARALLEL COMPUTATIONS 79

3.1 Ideal Parallel Computation 79

3.2 Embarrassingly Parallel Examples 81
Geometrical Transformations of Images 81
Mandelbrot Set 86
Monte Carlo Methods 93

3.3 Summary 98

Further Reading 99

Bibliography 99

Problems 100

CHAPTER 4 PARTITIONING AND DIVIDE-AND-CONQUER STRATEGIES 106

4.1 Partitioning 106
Partitioning Strategies 106
Divide and Conquer 111
M-*ary Divide and Conquer 116*

4.2 Partitioning and Divide-and-Conquer Examples 117
Sorting Using Bucket Sort 117
Numerical Integration 122
N-*Body Problem 126*

4.3 Summary 131

Further Reading 131

Bibliography 132

Problems 133

CHAPTER 5 PIPELINED COMPUTATIONS 140

5.1 Pipeline Technique 140

5.2 Computing Platform for Pipelined Applications 144

5.3 Pipeline Program Examples 145
Adding Numbers 145
Sorting Numbers 148
Prime Number Generation 152
Solving a System of Linear Equations — Special Case 154

5.4 Summary 157

Further Reading 158

Bibliography 158

Problems 158

CHAPTER 6 SYNCHRONOUS COMPUTATIONS 163

6.1 Synchronization 163
Barrier 163
Counter Implementation 165
Tree Implementation 167
Butterfly Barrier 167
Local Synchronization 169
Deadlock 169

6.2 Synchronized Computations 170
Data Parallel Computations 170
Synchronous Iteration 173

6.3 Synchronous Iteration Program Examples 174
Solving a System of Linear Equations by Iteration 174
Heat-Distribution Problem 180
Cellular Automata 190

6.4 Partially Synchronous Methods 191

6.5 Summary 193

Further Reading 193

Bibliography 193

Problems 194

CHAPTER 7 LOAD BALANCING AND TERMINATION DETECTION 201

7.1 Load Balancing 201

7.2 Dynamic Load Balancing 203
Centralized Dynamic Load Balancing 204
Decentralized Dynamic Load Balancing 205
Load Balancing Using a Line Structure 207

7.3 Distributed Termination Detection Algorithms 210
Termination Conditions 210
Using Acknowledgment Messages 211
Ring Termination Algorithms 212
Fixed Energy Distributed Termination Algorithm 214

7.4 Program Example 214
Shortest-Path Problem 214
Graph Representation 215
Searching a Graph 217

7.5 Summary 223

Further Reading 223

Bibliography 224

Problems 225

CHAPTER 8 PROGRAMMING WITH SHARED MEMORY 230

8.1 Shared Memory Multiprocessors 230

8.2 Constructs for Specifying Parallelism 232
Creating Concurrent Processes 232
Threads 234

8.3 Sharing Data 239
Creating Shared Data 239
Accessing Shared Data 239

8.4 Parallel Programming Languages and Constructs 247
Languages 247
Language Constructs 248
Dependency Analysis 250

8.5 OpenMP 253

8.6 Performance Issues 258
Shared Data Access 258
Shared Memory Synchronization 260
Sequential Consistency 262

8.7 Program Examples 265
UNIX Processes 265
Pthreads Example 268
Java Example 270

8.8 Summary 271

Further Reading 272

Bibliography 272

Problems 273

CHAPTER 9 DISTRIBUTED SHARED MEMORY SYSTEMS AND PROGRAMMING 279

9.1 Distributed Shared Memory 279

9.2 Implementing Distributed Shared Memory 281
Software DSM Systems 281
Hardware DSM Implementation 282
Managing Shared Data 283
Multiple Reader/Single Writer Policy in a Page-Based System 284

9.3 Achieving Consistent Memory in a DSM System 284

9.4 Distributed Shared Memory Programming Primitives 286
Process Creation 286
Shared Data Creation 287
Shared Data Access 287
Synchronization Accesses 288
Features to Improve Performance 288

9.5 Distributed Shared Memory Programming 290

9.6 Implementing a Simple DSM system 291
User Interface Using Classes and Methods 291
Basic Shared-Variable Implementation 292
Overlapping Data Groups 295

9.7 Summary 297

Further Reading 297

Bibliography 297

Problems 298

PART II ALGORITHMS AND APPLICATIONS 301

CHAPTER 10 SORTING ALGORITHMS 303

10.1 General 303
Sorting 303
Potential Speedup 304

10.2 Compare-and-Exchange Sorting Algorithms 304
Compare and Exchange 304
Bubble Sort and Odd-Even Transposition Sort 307
Mergesort 311
Quicksort 313
Odd-Even Mergesort 316
Bitonic Mergesort 317

10.3 Sorting on Specific Networks 320
Two-Dimensional Sorting 321
Quicksort on a Hypercube 323

10.4 Other Sorting Algorithms 327
Rank Sort 327
Counting Sort 330
Radix Sort 331
Sample Sort 333
Implementing Sorting Algorithms on Clusters 333

10.5 Summary 335

Further Reading 335

Bibliography 336

Problems 337

CHAPTER 11 NUMERICAL ALGORITHMS 340

11.1 Matrices — A Review 340
Matrix Addition 340
Matrix Multiplication 341
Matrix-Vector Multiplication 341
Relationship of Matrices to Linear Equations 342

11.2 Implementing Matrix Multiplication 342
Algorithm 342
Direct Implementation 343
Recursive Implementation 346
Mesh Implementation 348
Other Matrix Multiplication Methods 352

11.3 Solving a System of Linear Equations 352
Linear Equations 352
Gaussian Elimination 353
Parallel Implementation 354

11.4 Iterative Methods 356
Jacobi Iteration 357
Faster Convergence Methods 360

11.5 Summary 365

Further Reading 365

Bibliography 365

Problems 366

CHAPTER 12 IMAGE PROCESSING 370

12.1 Low-level Image Processing 370

12.2 Point Processing 372

12.3 Histogram 373

12.4 Smoothing, Sharpening, and Noise Reduction 374
Mean 374
Median 375
Weighted Masks 377

12.5 Edge Detection 379
Gradient and Magnitude 379
Edge-Detection Masks 380

12.6 The Hough Transform 383

12.7 Transformation into the Frequency Domain 387
Fourier Series 387
Fourier Transform 388
Fourier Transforms in Image Processing 389
Parallelizing the Discrete Fourier Transform Algorithm 391
Fast Fourier Transform 395

12.8 Summary 400

Further Reading 401

Bibliography 401

Problems 403

CHAPTER 13 SEARCHING AND OPTIMIZATION 406

13.1 Applications and Techniques 406

13.2 Branch-and-Bound Search 407
Sequential Branch and Bound 407
Parallel Branch and Bound 409

13.3 Genetic Algorithms 411
Evolution and Genetic Algorithms 411
Sequential Genetic Algorithms 413
Initial Population 413
Selection Process 415
Offspring Production 416
Variations 418
Termination Conditions 418
Parallel Genetic Algorithms 419

13.4 Successive Refinement 423

13.5 Hill Climbing 424
Banking Application 425
Hill Climbing in a Banking Application 427
Parallelization 428

13.6 Summary 428

Further Reading 428

Bibliography 429

Problems 430

APPENDIX A BASIC MPI ROUTINES 437

APPENDIX B BASIC PTHREAD ROUTINES 444

APPENDIX C OPENMP DIRECTIVES, LIBRARY FUNCTIONS, AND ENVIRONMENT VARIABLES 449

INDEX 460

Chapter 1

Parallel Computers

In this chapter, we describe the demand for greater computational power from computers and the concept of using computers with multiple internal processors and multiple interconnected computers. The prospects for increased speed of execution by using multiple computers or multiple processors and the limitations are discussed. Then, the various ways that such systems can be constructed are described, in particular by using multiple computers in a cluster, which has become a very cost-effective computer platform for high-performance computing.

1.1 THE DEMAND FOR COMPUTATIONAL SPEED

There is a continual demand for greater computational power from computer systems than is currently possible. Areas requiring great computational speed include numerical simulation of scientific and engineering problems. Such problems often need huge quantities of repetitive calculations on large amounts of data to give valid results. Computations must be completed within a "reasonable" time period. In the manufacturing realm, engineering calculations and simulations must be achieved within seconds or minutes if possible. A simulation that takes two weeks to reach a solution is usually unacceptable in a design environment, because the time has to be short enough for the designer to work effectively. As systems become more complex, it takes increasingly more time to simulate them. There are some problems that have a specific deadline for the computations, for example weather forecasting. Taking two days to forecast the local weather accurately for the next day would make the prediction useless. Some areas, such as modeling large DNA structures and global weather forecasting, are *grand challenge problems*. A grand challenge problem is one that cannot be solved in a reasonable amount of time with today's computers.

Weather forecasting by computer (*numerical weather prediction*) is a widely quoted example that requires very powerful computers. The atmosphere is modeled by dividing it into three-dimensional regions or cells. Rather complex mathematical equations are used to capture the various atmospheric effects. In essence, conditions in each cell (temperature, pressure, humidity, wind speed and direction, etc.) are computed at time intervals using conditions existing in the previous time interval in the cell and nearby cells. The calculations of each cell are repeated many times to model the passage of time. The key feature that makes the simulation significant is the number of cells that are necessary. For forecasting over days, the atmosphere is affected by very distant events, and thus a large region is necessary. Suppose we consider the whole global atmosphere divided into cells of size 1 mile × 1 mile × 1 mile to a height of 10 miles (10 cells high). A rough calculation leads to about 5×10^8 cells. Suppose each calculation requires 200 floating-point operations (the type of operation necessary if the numbers have a fractional part or are raised to a power). In one time step, 10^{11} floating point operations are necessary. If we were to forecast the weather over seven days using 1-minute intervals, there would be 10^4 time steps and 10^{15} floating-point operations in total. A computer capable of 1 Gflops (10^9 floating-point operations/sec) with this calculation would take 10^6 seconds or over 10 days to perform the calculation. To perform the calculation in 5 minutes would require a computer operating at 3.4 Tflops (3.4×10^{12} floating-point operations/sec).

Another problem that requires a huge number of calculations is predicting the motion of the astronomical bodies in space. Each body is attracted to each other body by gravitational forces. These are long-range forces that can be calculated by a simple formula (see Chapter 4). The movement of each body can be predicted by calculating the total force experienced by the body. If there are N bodies, there will be $N - 1$ forces to calculate for each body, or approximately N^2 calculations, in total. After the new positions of the bodies are determined, the calculations must be repeated. A snapshot of an undergraduate student's results for this problem, given as a programming assignment with a few bodies, is shown in Figure 1.1. However, there could be a huge number of bodies to consider. A galaxy might have, say, 10^{11} stars. This suggests that 10^{22} calculations have to be repeated. Even using the efficient approximate algorithm described in Chapter 4, which requires $N \log_2 N$ calculations (but more involved calculations), the number of calculations is still enormous ($10^{11} \log_2 10^{11}$). It would require significant time on a single-processor system. Even if each calculation could be done in 1 μs (10^{-6} seconds, an extremely optimistic figure, since it involves several multiplications and divisions), it would take 10^9 years for one iteration using the N^2 algorithm and almost a year for one iteration using the $N \log_2 N$ algorithm. The N-body problem also appears in modeling chemical and biological systems at the molecular level and takes enormous computational power.

Global weather forecasting and simulation of a large number of bodies (astronomical or molecular) are traditional examples of applications that require immense computational power, but it is human nature to continually envision new applications that exceed the capabilities of present-day computer systems and require more computational speed than available. Recent applications, such as virtual reality, require considerable computational speed to achieve results with images and movements that appear real without any jerking. It seems that whatever the computational speed of current processors, there will be applications that require still more computational power.

Figure 1.1 Astrophysical N-body simulation by Scott Linssen (undergraduate student, University of North Carolina at Charlotte).

A traditional computer has a single processor for performing the actions specified in a program. One way of increasing the computational speed, a way that has been considered for many years, is by using multiple processors within a single computer (multiprocessor) or alternatively multiple computers, operating together on a single problem. In either case, the overall problem is split into parts, each of which is performed by a separate processor in parallel. Writing programs for this form of computation is known as *parallel programming*. The computing platform, a *parallel computer*, could be a specially designed computer system containing multiple processors or several computers interconnected in some way. The approach should provide a significant increase in performance. The idea is that p processors/computers could provide up to p times the computational speed of a single processor/computer, no matter what the current speed of the processor/computer, with the expectation that the problem would be completed in $1/p$th of the time. Of course, this is an ideal situation that is rarely achieved in practice. Problems often cannot be divided perfectly into independent parts, and interaction is necessary between the parts, both for data transfer and synchronization of computations. However, substantial improvement can be achieved, depending upon the problem and the amount of parallelism in the problem. What makes parallel computing timeless is that the continual improvements in the execution speed of processors simply make parallel computers even faster, and there will always be grand challenge problems that cannot be solved in a reasonable amount of time on current computers.

Apart from obtaining the potential for increased speed on an existing problem, the use of multiple computers/processors often allows a larger problem or a more precise solution of a problem to be solved in a reasonable amount of time. For example, computing many physical phenomena involves dividing the problem into discrete solution points. As we have mentioned, forecasting the weather involves dividing the air into a three-dimensional grid of solution points. Two- and three-dimensional grids of solution points occur in many other applications. A multiple computer or multiprocessor solution will often allow more solution

points to be computed in a given time, and hence a more precise solution. A related factor is that multiple computers very often have more total main memory than a single computer, enabling problems that require larger amounts of main memory to be tackled.

Even if a problem can be solved in a reasonable time, situations arise when the same problem has to be evaluated multiple times with different input values. This situation is especially applicable to parallel computers, since without any alteration to the program, multiple instances of the same program can be executed on different processors/computers simultaneously. Simulation exercises often come under this category. The simulation code is simply executed on separate computers simultaneously but with different input values.

Finally, the emergence of the Internet and the World Wide Web has spawned a new area for parallel computers. For example, Web servers must often handle thousands of requests per hour from users. A multiprocessor computer, or more likely nowadays multiple computers connected together as a "cluster," are used to service the requests. Individual requests are serviced by different processors or computers simultaneously. On-line banking and on-line retailers all use clusters of computers to service their clients.

The parallel computer is not a new idea; in fact it is a very old idea. For example, Gill wrote about parallel programming in 1958 (Gill, 1958). Holland wrote about a "computer capable of executing an arbitrary number of sub-programs simultaneously" in 1959 (Holland, 1959). Conway described the design of a parallel computer and its programming in 1963 (Conway, 1963). Notwithstanding the long history, Flynn and Rudd (1996) write that "the continued drive for higher- and higher-performance systems . . . leads us to one simple conclusion: the future is parallel." We concur.

1.2 POTENTIAL FOR INCREASED COMPUTATIONAL SPEED

In the following and in subsequent chapters, the number of processes or processors will be identified as p. We will use the term "*multiprocessor*" to include all parallel computer systems that contain more than one processor.

1.2.1 Speedup Factor

Perhaps the first point of interest when developing solutions on a multiprocessor is the question of how much faster the multiprocessor solves the problem under consideration. In doing this comparison, one would use the best solution on the single processor, that is, the best sequential algorithm on the single-processor system to compare against the parallel algorithm under investigation on the multiprocessor. The *speedup factor*, $S(p)$,[1] is a measure of relative performance, which is defined as:

$$S(p) = \frac{\text{Execution time using single processor system (with the best sequential algorithm)}}{\text{Execution time using a multiprocessor with } p \text{ processors}}$$

We shall use t_s as the execution time of the best sequential algorithm running on a single processor and t_p as the execution time for solving the same problem on a multiprocessor.

[1] The speedup factor is normally a function of both p and the number of data items being processed, n, i.e. $S(p,n)$. We will introduce the number of data items later. At this point, the only variable is p.

Then:

$$S(p) = \frac{t_s}{t_p}$$

$S(p)$ gives the increase in speed in using the multiprocessor. Note that the underlying algorithm for the parallel implementation might not be the same as the algorithm on the single-processor system (and is usually different).

In a theoretical analysis, the speedup factor can also be cast in terms of computational steps:

$$S(p) = \frac{\text{Number of computational steps using one processor}}{\text{Number of parallel computational steps with } p \text{ processors}}$$

For sequential computations, it is common to compare different algorithms using time complexity, which we will review in Chapter 2. Time complexity can be extended to parallel algorithms and applied to the speedup factor, as we shall see. However, considering computational steps alone may not be useful, as parallel implementations may require expense communications between the parallel parts, which is usually much more time-consuming than computational steps. We shall look at this in Chapter 2.

The maximum speedup possible is usually p with p processors (*linear speedup*). The speedup of p would be achieved when the computation can be divided into equal-duration processes, with one process mapped onto one processor and no additional overhead in the parallel solution.

$$S(p) \leq \frac{t_s}{t_s/p} = p$$

Superlinear speedup, where $S(p) > p$, may be seen on occasion, but usually this is due to using a suboptimal sequential algorithm, a unique feature of the system architecture that favors the parallel formation, or an indeterminate nature of the algorithm. Generally, if a purely deterministic parallel algorithm were to achieve better than p times the speedup over the current sequential algorithm, the parallel algorithm could be emulated on a single processor one parallel part after another, which would suggest that the original sequential algorithm was not optimal.

One common reason for superlinear speedup is extra memory in the multiprocessor system. For example, suppose the main memory associated with each processor in the multiprocessor system is the same as that associated with the processor in a single-processor system. Then, the total main memory in the multiprocessor system is larger than that in the single-processor system, and can hold more of the problem data at any instant, which leads to less disk memory traffic.

Efficiency. It is sometimes useful to know how long processors are being used on the computation, which can be found from the (system) *efficiency*. The efficiency, E, is defined as

$$E = \frac{\text{Execution time using one processor}}{\text{Execution time using a multiprocessor} \times \text{number of processors}}$$

$$= \frac{t_s}{t_p \times p}$$

which leads to

$$E = \frac{S(p)}{p} \times 100\%$$

when E is given as a percentage. For example, if $E = 50\%$, the processors are being used half the time on the actual computation, on average. The efficiency of 100% occurs when all the processors are being used on the computation at all times and the speedup factor, $S(p)$, is p.

1.2.2 What Is the Maximum Speedup?

Several factors will appear as overhead in the parallel version and limit the speedup, notably

1. Periods when not all the processors can be performing useful work and are simply idle.
2. Extra computations in the parallel version not appearing in the sequential version; for example, to recompute constants locally.
3. Communication time between processes.

It is reasonable to expect that some part of a computation cannot be divided into concurrent processes and must be performed sequentially. Let us assume that during some period, perhaps an initialization period or the period before concurrent processes are set up, only one processor is doing useful work, and for the rest of the computation additional processors are operating on processes.

Assuming there will be some parts that are only executed on one processor, the ideal situation would be for all the available processors to operate simultaneously for the other times. If the fraction of the computation that cannot be divided into concurrent tasks is f, and no overhead is incurred when the computation is divided into concurrent parts, the time to perform the computation with p processors is given by $ft_s + (1 - f)t_s/p$, as illustrated in Figure 1.2. Illustrated is the case with a single serial part at the beginning of the computation, but the serial part could be distributed throughout the computation. Hence, the speedup factor is given by

$$S(p) = \frac{t_s}{ft_s + (1-f)t_s/p} = \frac{p}{1 + (p-1)f}$$

This equation is known as *Amdahl's law* (Amdahl, 1967). Figure 1.3 shows $S(p)$ plotted against number of processors and against f. We see that indeed a speed improvement is indicated. However, the fraction of the computation that is executed by concurrent processes needs to be a substantial fraction of the overall computation if a significant increase in speed is to be achieved. Even with an infinite number of processors, the maximum speedup is limited to $1/f$; i.e.,

$$\underset{p \to \infty}{S(p)} = \frac{1}{f}$$

For example, with only 5% of the computation being serial, the maximum speedup is 20, irrespective of the number of processors. Amdahl used this argument to promote single-processor

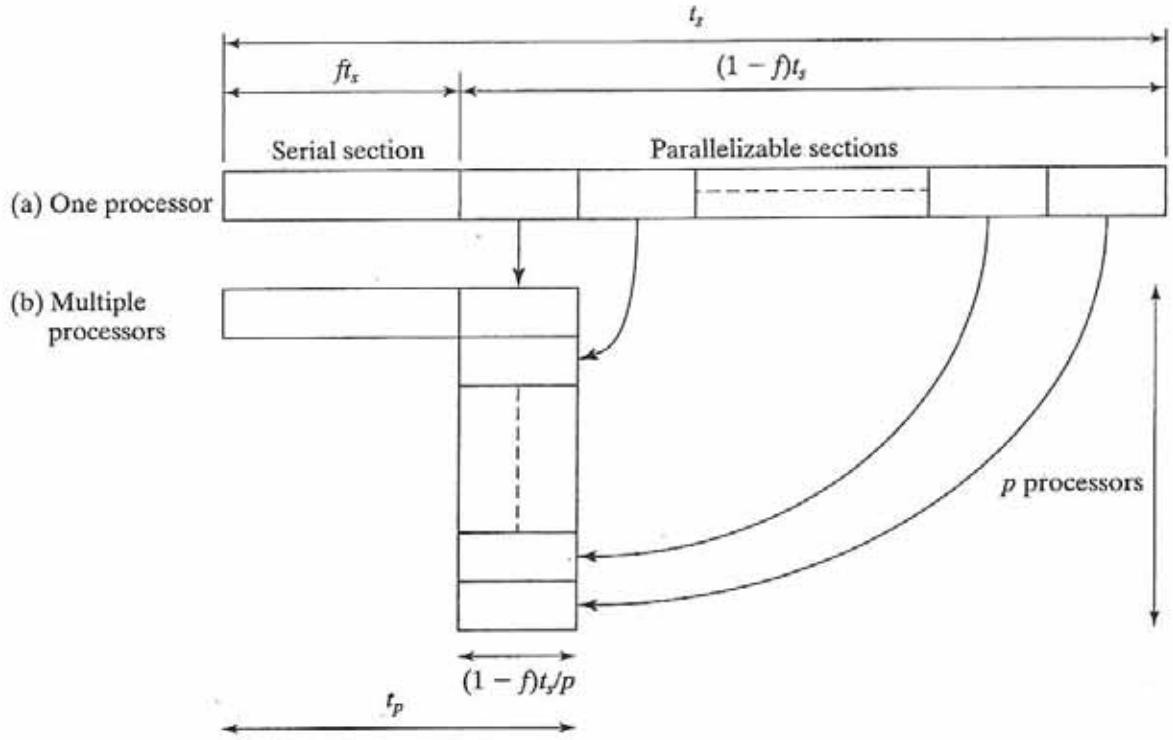


Figure 1.2 Parallelizing sequential problem — Amdahl's law. pessimistic

systems in the 1960s. Of course, one can counter this by saying that even a speedup of 20 would be impressive.

Orders-of-magnitude improvements are possible in certain circumstances. For example, superlinear speedup can occur in search algorithms. In search problems performed by exhaustively looking for the solution, suppose the solution space is divided among the processors for each one to perform an independent search. In a sequential

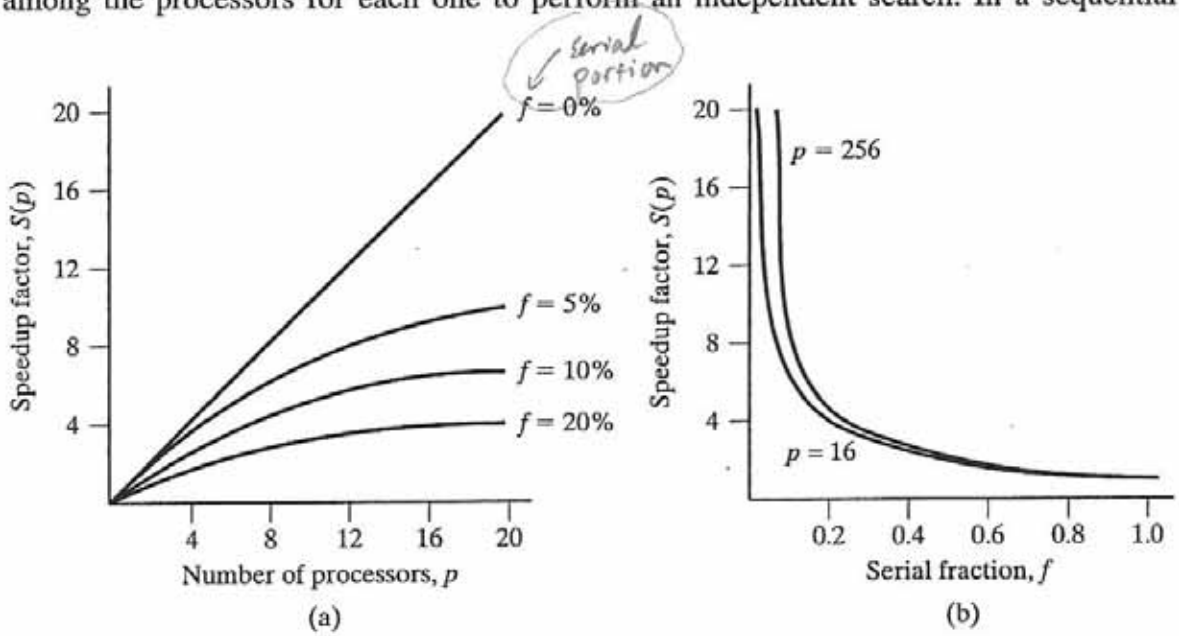
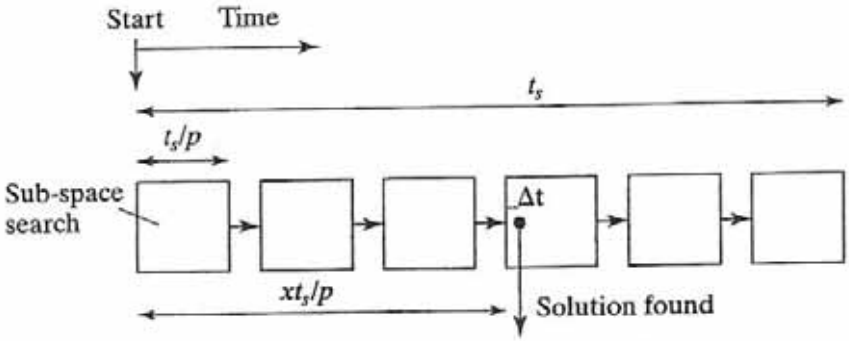


Figure 1.3 (a) Speedup against number of processors. (b) Speedup against serial fraction, f.

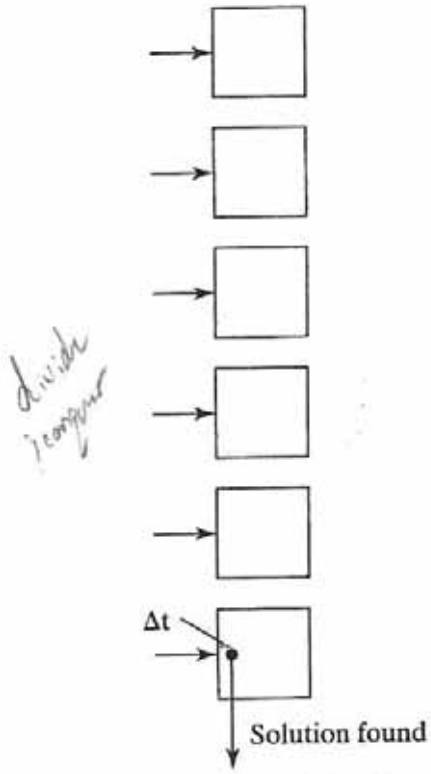
implementation, the different search spaces are attacked one after the other. In parallel implementation, they can be done simultaneously, and one processor might find the solution almost immediately. In the sequential version, suppose x sub-spaces are searched and then the solution is found in time Δt in the next sub-space search. The number of previously searched sub-spaces, say x, is indeterminate and will depend upon the problem. In the parallel version, the solution is found immediately in time Δt, as illustrated in Figure 1.4.

The speedup is then given by

$$S(p) = \frac{\left(x \times \frac{t_s}{p}\right) + \Delta t}{\Delta t}$$



(a) Searching each sub-space sequentially



(b) Searching each sub-space in parallel

Figure 1.4 Superlinear speedup.

The worst case for the sequential search is when the solution is found in the last sub-space search, and the parallel version offers the greatest benefit:

$$S(p) = \frac{\left(\frac{p-1}{p}\right) \times t_s + \Delta t}{\Delta t} \rightarrow \infty \quad \text{as } \Delta t \text{ tends to zero}$$

The least advantage for the parallel version would be when the solution is found in the first sub-space search of the sequential search:

$$S(p) = \frac{\Delta t}{\Delta t} = 1$$

The actual speedup will depend upon which sub-space holds the solution but could be extremely large.

Scalability. The performance of a system will depend upon the size of the system, i.e., the number of processors, and generally the larger the system the better, but this comes with a cost. *Scalability* is a rather imprecise term. It is used to indicate a hardware design that allows the system to be increased in size and in doing so to obtain increased performance. This could be described as *architecture* or *hardware scalability.* Scalability is also used to indicate that a parallel algorithm can accommodate increased data items with a low and bounded increase in computational steps. This could be described as *algorithmic scalability.*

Of course, we would want all multiprocessor systems to be *architecturally scalable* (and manufacturers will market their systems as such), but this will depend heavily upon the design of the system. Usually, as we add processors to a system, the interconnection network must be expanded. Greater communication delays and increased contention results, and the system efficiency, E, reduces. The underlying goal of most multiprocessor designs is to achieve scalability, and this is reflected in the multitude of interconnection networks that have been devised.

Combined architecture/algorithmic scalability suggests that increased problem size can be accommodated with increased system size for a particular architecture and algorithm. Whereas increasing the size of the system clearly means adding processors, increasing the size of the problem requires clarification. Intuitively, we would think of the number of data elements being processed in the algorithm as a measure of size. However, doubling the problem size would not necessarily double the number of computational steps. It will depend upon the problem. For example, adding two matrices, as discussed in Chapter 11, has this effect, but multiplying matrices does not. The number of computational steps for multiplying matrices quadruples. Hence, scaling different problems would imply different computational requirements. An alternative definition of *problem size* is to equate problem size with the number of basic steps in the best sequential algorithm. Of course, even with this definition, if we increase the number of data points, we will increase the problem size.

In subsequent chapters, in addition to number of processors, p, we will also use n as the number of input data elements in a problem.[2] These two, p and n, usually can be altered in an attempt to improve performance. Altering p alters the size of the computer system,

and altering n alters the size of the problem. Usually, increasing the problem size improves the relative performance because more parallelism can be achieved.

Gustafson presented an argument based upon scalability concepts to show that Amdahl's law was not as significant as first supposed in determining the potential speedup limits (Gustafson, 1988). Gustafson attributed formulating the idea into an equation to E. Barsis. Gustafson makes the observation that in practice a larger multiprocessor usually allows a larger-size problem to be undertaken in a reasonable execution time. Hence in practice, the problem size selected frequently depends of the number of available processors. Rather than assume that the problem size is fixed, it is just as valid to assume that the parallel execution time is fixed. As the system size is increased (p increased), the problem size is increased to maintain constant parallel-execution time. In increasing the problem size, Gustafson also makes the case that the serial section of the code is normally fixed and does not increase with the problem size.

Using the constant parallel-execution time constraint, the resulting speedup factor will be numerically different from Amdahl's speedup factor and is called a *scaled speedup factor* (i.e, the speedup factor when the problem is scaled). For Gustafson's scaled speedup factor, the parallel execution time, t_p, is constant rather than the serial execution time, t_s, in Amdahl's law. For the derivation of Gustafson's law, we shall use the same terms as for deriving Amdahl's law, but it is necessary to separate out the serial and parallelizable sections of the sequential execution time, t_s, into $ft_s + (1-f)t_s$ as the serial section ft_s is a constant. For algebraic convenience, let the parallel execution time, $t_p = ft_s + (1-f)t_s/p = 1$. Then, with a little algebraic manipulation, the serial execution time, t_s, becomes $ft_s + (1-f)t_s = p + (1-p)ft_s$. The scaled speedup factor then becomes

$$S_s(p) = \frac{ft_s + (1-f)t_s}{ft_s + (1-f)t_s/p} = \frac{p + (1-p)ft_s}{1} = p + (1-p)ft_s$$

which is called *Gustafson's law*. There are two assumptions in this equation: the parallel execution time is constant, and the part that must be executed sequentially, ft_s, is also constant and not a function of p. Gustafson's observation here is that the scaled speedup factor is a line of negative slope $(1-p)$ rather than the rapid reduction previously illustrated in Figure 1.3(b). For example, suppose we had a serial section of 5% and 20 processors; the speedup is $0.05 + 0.95(20) = 19.05$ according to the formula instead of 10.26 according to Amdahl's law. (Note, however, the different assumptions.) Gustafson quotes examples of speedup factors of 1021, 1020, and 1016 that have been achieved in practice with a 1024-processor system on numerical and simulation problems.

Apart from constant problem size scaling (Amdahl's assumption) and time-constrained scaling (Gustafson's assumption), scaling could be memory-constrained scaling. In memory-constrained scaling, the problem is scaled to fit in the available memory. As the number of processors grows, normally the memory grows in proportion. This form can lead to significant increases in the execution time (Singh, Hennessy, and Gupta, 1993).

[2] For matrices, we consider $n \times n$ matrices.

1.2.3 Message-Passing Computations

The analysis so far does not take account of message-passing, which can be a very significant overhead in the computation in message-passing programming. In this form of parallel programming, messages are sent between processes to pass data and for synchronization purposes. Thus,

$$t_p = t_{comm} + t_{comp}$$

where t_{comm} is the communication time, and t_{comp} is the computation time. As we divide the problem into parallel parts, the computation time of the parallel parts generally decreases because the parts become smaller, and the communication time between the parts generally increases (as there are more parts communicating). At some point, the communication time will dominate the overall execution time and the parallel execution time will actually increase. It is essential to reduce the communication overhead because of the significant time taken by interprocessor communication. The communication aspect of the parallel solution is usually not present in the sequential solution and considered as an overhead.

The ratio

$$\text{Computation/communication ratio} = \frac{\text{Computation time}}{\text{Communication time}} = \frac{t_{comp}}{t_{comm}}$$

can be used as a metric. In subsequent chapters, we will develop equations for the computation time and the communication time in terms of number of processors (p) and number of data elements (n) for algorithms and problems under consideration to get a handle on the potential speedup possible and effect of increasing p and n.

In a practical situation we may not have much control over the value of p, that is, the size of the system we can use (except that we could map more than one process of the problem onto one processor, although this is not usually beneficial). Suppose, for example, that for some value of p, a problem requires $c_1 n$ computations and $c_2 n^2$ communications. Clearly, as n increases, the communication time increases faster than the computation time. This can be seen clearly from the computation/communication ratio, $(c_1/c_2 n)$, which can be cast in time-complexity notation to remove constants (see Chapter 2). Usually, we want the computation/communication ratio to be as high as possible, that is, some highly increasing function of n so that increasing the problem size lessens the effects of the communication time. Of course, this is a complex matter with many factors. Finally, one can only verify the execution speed by executing the program on a real multiprocessor system, and it is assumed this would then be done. Ways of measuring the actual execution time are described in the next chapter.

1.3 TYPES OF PARALLEL COMPUTERS

Having convinced ourselves that there is potential for speedup with the use of multiple processors or computers, let us explore how a multiprocessor or multicomputer could be constructed. A parallel computer, as we have mentioned, is either a single computer with multiple internal processors or multiple computers interconnected to form a coherent

high-performance computing platform. In this section, we shall look at specially designed parallel computers, and later in the chapter we will look at using an off-the-shelf "commodity" computer configured as a cluster. The term *parallel computer* is usually reserved for specially designed components. There are two basic types of parallel computer:

1. Shared memory multiprocessor
2. Distributed-memory multicomputer.

1.3.1 Shared Memory Multiprocessor System

A conventional computer consists of a processor executing a program stored in a (main) memory, as shown in Figure 1.5. Each main memory location in the memory is located by a number called its *address*. Addresses start at 0 and extend to $2^b - 1$ when there are b bits (binary digits) in the address.

A natural way to extend the single-processor model is to have multiple processors connected to multiple memory modules, such that each processor can access any memory module in a so–called *shared memory* configuration, as shown in Figure 1.6. The connection between the processors and memory is through some form of *interconnection network*. A shared memory multiprocessor system employs a *single address space*, which means that each location in the whole main memory system has a unique address that is used by each processor to access the location. Although not shown in these "models," real systems have high-speed cache memory, which we shall discuss later.

Programming a shared memory multiprocessor involves having executable code stored in the shared memory for each processor to execute. The data for each program will also be stored in the shared memory, and thus each program could access all the data if

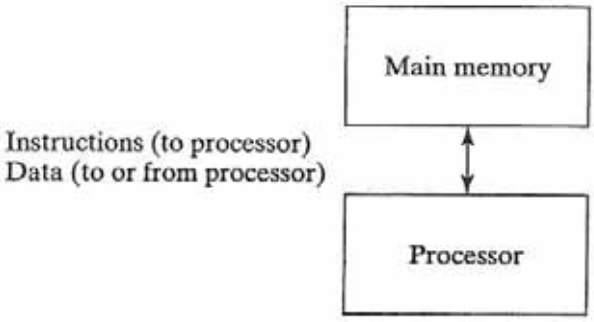


Figure 1.5 Conventional computer having a single processor and memory.

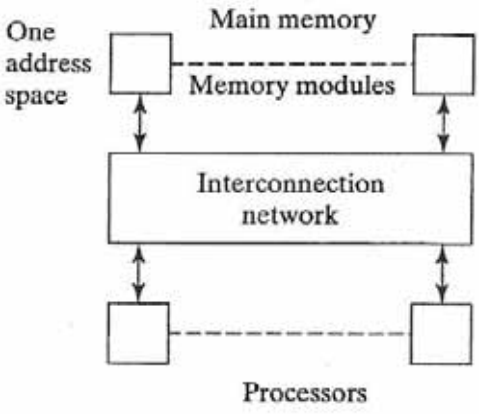


Figure 1.6 Traditional shared memory multiprocessor model.

needed. A programmer can create the executable code and shared data for the processors in different ways, but the final result is to have each processor execute its own program or code sequences from the shared memory. (Typically, all processors execute the same program.)

One way for the programmer to produce the executable code for each processor is to use a high-level parallel programming language that has special parallel programming constructs and statements for declaring shared variables and parallel code sections. The compiler is responsible for producing the final executable code from the programmer's specification in the program. However, a completely new parallel programming language would not be popular with programmers. More likely when using a compiler to generate parallel code from the programmer's "source code," a regular sequential programming language would be used with preprocessor directives to specify the parallelism. An example of this approach is OpenMP (Chandra et al., 2001), an industry-standard set of compiler directives and constructs added to C/C++ and Fortran. Alternatively, so-called *threads* can be used that contain regular high-level language code sequences for individual processors. These code sequences can then access shared locations. Another way that has been explored over the years, and is still finding interest, is to use a regular sequential programming language and modify the syntax to specify parallelism. A recent example of this approach is UPC (Unified Parallel C) (see http://upc.gwu.edu). More details on exactly how to program shared memory systems using threads and other ways are given in Chapter 8.

From a programmer's viewpoint, the shared memory multiprocessor is attractive because of the convenience of sharing data. Small (two-processor and four-processor) shared memory multiprocessor systems based upon a bus interconnection structure-as illustrated in Figure 1.7 are common; for example dual-Pentium® and quad-Pentium systems. Two-processor shared memory systems are particularly cost-effective. However, it is very difficult to implement the hardware to achieve fast access to all the shared memory by all the processors with a large number of processors. Hence, most large shared memory systems have some form of hierarchical or distributed memory structure. Then, processors can physically access nearby memory locations much faster than more distant memory locations. The term *nonuniform memory access* (NUMA) is used in these cases, as opposed to *uniform memory access* (UMA).

Conventional single processors have fast cache memory to hold copies of recently referenced memory locations, thus reducing the need to access the main memory on every memory reference. Often, there are two levels of cache memory between the processor and the main memory. Cache memory is carried over into shared memory multiprocessors by providing each processor with its own local cache memory. Fast local cache memory with each processor can somewhat alleviate the problem of different access times to different main memories in larger systems, but making sure that copies of the same data in different

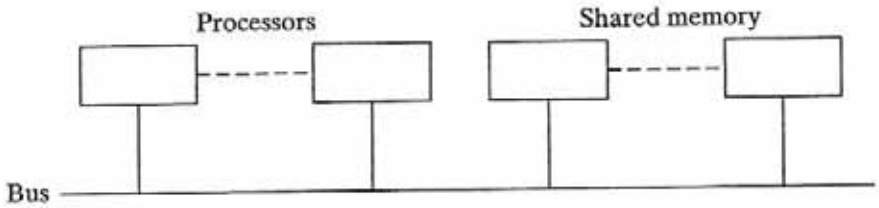


Figure 1.7 Simplistic view of a small shared memory multiprocessor.

caches are identical becomes a complex issue that must be addressed. One processor writing to a cached data item often requires all the other copies of the cached item in the system to be made invalid. Such matters are briefly covered in Chapter 8.

1.3.2 Message-Passing Multicomputer

An alternative form of multiprocessor to a shared memory multiprocessor can be created by connecting complete computers through an interconnection network, as shown in Figure 1.8. Each computer consists of a processor and local memory but this memory is not accessible by other processors. The interconnection network provides for processors to send messages to other processors. The messages carry data from one processor to another as dictated by the program. Such multiprocessor systems are usually called *message-passing multiprocessors*, or simply *multicomputers*, especially if they consist of self-contained computers that could operate separately.

Programming a message-passing multicomputer still involves dividing the problem into parts that are intended to be executed simultaneously to solve the problem. Programming could use a parallel or extended sequential language, but a common approach is to use message-passing library routines that are inserted into a conventional sequential program for message passing. Often, we talk in terms of *processes*. A problem is divided into a number of concurrent processes that may be executed on a different computer. If there were six processes and six computers, we might have one process executed on each computer. If there were more processes than computers, more than one process would be executed on one computer, in a time-shared fashion. Processes communicate by sending messages; this will be the only way to distribute data and results between processes.

The message-passing multicomputer will physically *scale* more easily than a shared memory multiprocessor. That is, it can more easily be made larger. There have been examples of specially designed message-passing processors. Message-passing systems can also employ general-purpose microprocessors.

Networks for Multicomputers. The purpose of the interconnection network shown in Figure 1.8 is to provide a physical path for messages sent from one computer to another computer. Key issues in network design are the *bandwidth*, *latency*, and *cost*. Ease of construction is also important. The *bandwidth* is the number of bits that can be transmitted in unit time, given as bits/sec. The *network latency* is the time to make a message transfer through the network. The *communication latency* is the total time to send the

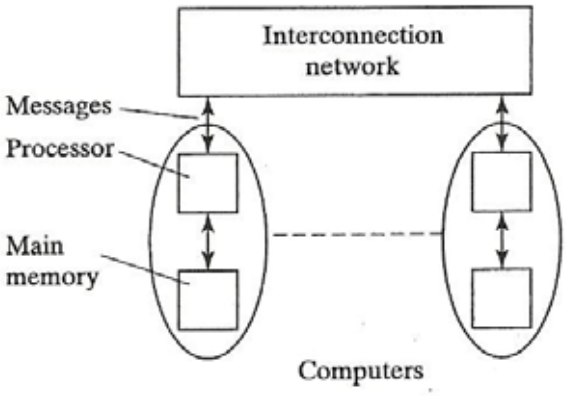


Figure 1.8 Message-passing multiprocessor model (multicomputer).

message, including the software overhead and interface delays. *Message latency*, or *startup time*, is the time to send a zero-length message, which is essentially the software and hardware overhead in sending a message (finding the route, packing, unpacking, etc.) onto which must be added the actual time to send the data along the interconnection path.

The number of physical links in a path between two nodes is an important consideration because it will be a major factor in determining the delay for a message. The *diameter* is the minimum number of links between the two farthest nodes (computers) in the network. Only the shortest routes are considered. How efficiently a parallel problem can be solved using a multicomputer with a specific network is extremely important. The diameter of the network gives the maximum distance that a single message must travel and can be used to find the communication lower bound of some parallel algorithms.

The *bisection width* of a network is the minimum number of links (or sometimes wires) that must be cut to divide the network into two equal parts. The *bisection bandwidth* is the collective bandwidth over these links, that is, the maximum number of bits that can be transmitted from one part of the divided network to the other part in unit time. These factor can also be important in evaluating parallel algorithms. Parallel algorithms usually require numbers to be moved about the network. To move numbers across the network from one side to the other we must use the links between the two halves, and the bisection width gives us the number of links available.

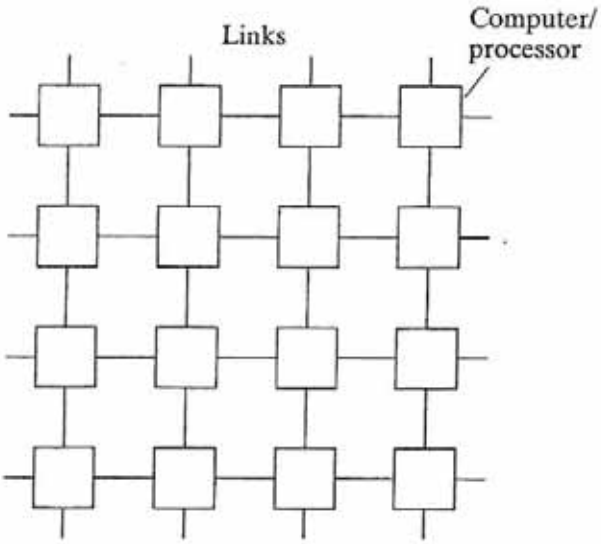
There are several ways one could interconnect computers to form a multicomputer system. For a very small system, one might consider connecting every computer to every other computer with links. With c computers, there are $c(c-1)/2$ links in all. Such exhaustive interconnections have application only for a very small system. For example, a set of four computers could reasonably be exhaustively interconnected. However, as the size increases, the number of interconnections clearly becomes impractical for economic and engineering reasons. Then we need to look at networks with restricted interconnection and switched interconnections.

There are two networks with restricted direct interconnections that have seen wide use — the *mesh* network and the *hypercube* network. Not only are these important as interconnection networks, the concepts also appear in the formation of parallel algorithms.

Mesh. A two-dimensional *mesh* can be created by having each node in a two-dimensional array connect to its four nearest neighbors, as shown in Figure 1.9. The diameter of a $\sqrt{p} \times \sqrt{p}$ mesh is $2(\sqrt{p}-1)$, since to reach one corner from the opposite corner requires a path to made across $(\sqrt{p}-1)$ nodes and down $(\sqrt{p}-1)$ nodes. The free ends of a mesh might circulate back to the opposite sides. Then the network is called a *torus*.

The mesh and torus networks are popular because of their ease of layout and expandability. If necessary, the network can be *folded*; that is, rows are interleaved and columns are interleaved so that the wraparound connections simply turn back through the network rather than stretch from one edge to the opposite edge. Three-dimensional meshes can be formed where each node connects to two nodes in the x-plane, the y-plane, and the z-plane. Meshes are particularly convenient for many scientific and engineering problems in which solution points are arranged in two-dimensional or three-dimensional arrays.

There have been several examples of message-passing multicomputer systems using two-dimensional or three-dimensional mesh networks, including the Intel Touchstone Delta computer (delivered in 1991, designed with a two-dimensional mesh), and the J-machine, a



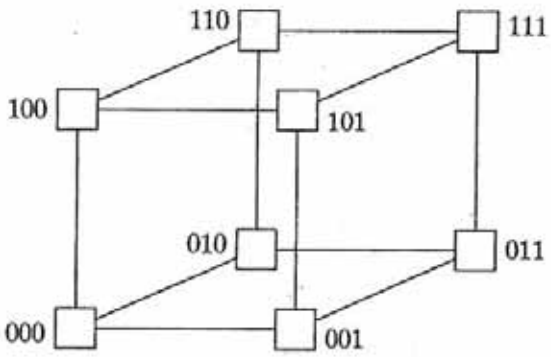
linear-bus

Figure 1.9 Two-dimensional array (mesh).

research prototype constructed at MIT in 1991 with a three-dimensional mesh. A more recent example of a system using a mesh is the ASCI Red supercomputer from the U.S. Department of Energy's Accelerated Strategic Computing Initiative, developed in 1995–97. ASCI Red, sited at Sandia National Laboratories, consists of 9,472 Pentium-II Xeon processors and uses a $38 \times 32 \times 2$ mesh interconnect for message passing. Meshes can also be used in shared memory systems.

Hypercube Network. In a d-dimensional (binary) *hypercube network*, each node connects to one node in each of the dimensions of the network. For example, in a three-dimensional hypercube, the connections in the x-direction, y-direction, and z-direction form a cube, as shown in Figure 1.10. Each node in a hypercube is assigned a d-bit binary address when there are d dimensions. Each bit is associated with one of the dimensions and can be a 0 or a 1, for the two nodes in that dimension. Nodes in a three-dimensional hypercube have a 3-bit address. Node 000 connects to nodes with addresses 001, 010, and 100. Node 111 connects to nodes 110, 101, and 011. Note that each node connects to nodes whose addresses differ by one bit. This characteristic can be extended for higher-dimension hypercubes. For example, in a five-dimensional hypercube, node 11101 connects to nodes 11100, 11111, 11001, 10101, and 01101.

A notable advantage of the hypercube is that the *diameter* of the network is given by $\log_2 p$ for a p-node hypercube, which has a reasonable (low) growth with increasing p. The



$8 = 2^3$

$64 = 2^6$

Figure 1.10 Three-dimensional hypercube.

number of links emanating from each node also only grows logarithmically. A very convenient aspect of the hypercube is the existence of a minimal distance deadlock-free routing algorithm. To describe this algorithm, let us route a message from a node X having a nodal address $X = x_{n-1}x_{n-2} \cdots x_1x_0$ to a destination node having a nodal address $Y = y_{n-1}y_{n-2} \cdots y_1y_0$. Each bit of Y that is different from that of X identifies one hypercube dimension that the route should take and can be found by performing the exclusive-OR function, $Z = X \oplus Y$, operating on pairs of bits. The dimensions to use in the routing are given by those bits of Z that are 1. At each node in the path, the exclusive-OR function between the current nodal address and the destination nodal address is performed. Usually the dimension identified by the most significant 1 in Z is chosen for the route. For example, the route taken from node 13 (001101) to node 42 (101010) in a six-dimensional hypercube would be node 13 (001101) to node 45 (101101) to node 41 (101001) to node 43 (101011) to node 42 (101010). This hypercube routing algorithm is sometimes called the *e-cube routing algorithm*, or *left-to-right routing*.

A d-dimensional hypercube actually consists of two $d-1$ dimensional hypercubes with dth dimension links between them. Figure 1.11 shows a four-dimensional hypercube drawn as two three-dimensional hypercubes with eight connections between them. Hence, the bisection width is 8. (The bisection width is $p/2$ for a p-node hypercube.) A five-dimensional hypercube consists of two four-dimensional hypercubes with connections between them, and so forth for larger hypercubes. In a practical system, the network must be laid out in two or possibly three dimensions.

Hypercubes are a part of a larger family of k-ary d-cubes; however, it is only the binary hypercube (with $k = 2$) that is really important as a basis for multicomputer construction and for parallel algorithms. The hypercube network became popular for constructing message-passing multicomputers after the pioneering research system called the Cosmic Cube was constructed at Caltech in the early 1980s (Seitz, 1985). However, interest in hypercubes has waned since the late 1980s.

As an alternative to direct links between individual computers, switches can be used in various configurations to route the messages between the computers.

Crossbar switch. The crossbar switch provides exhaustive connections using one switch for each connection. It is employed in shared memory systems more so than

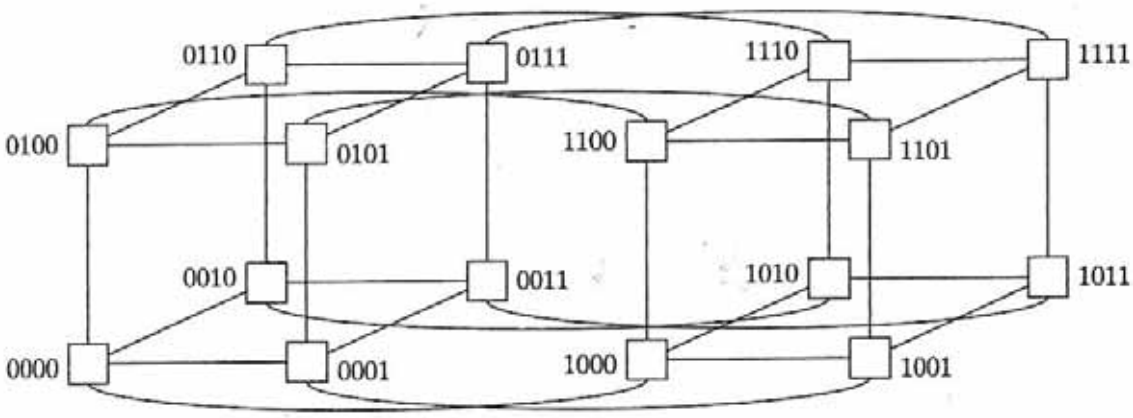


Figure 1.11 Four-dimensional hypercube.

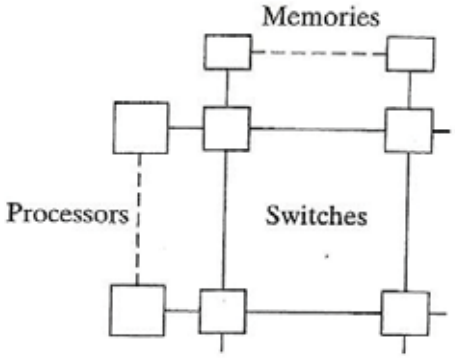


Figure 1.12 Cross-bar switch.

message-passing systems for connecting processor to memories. The layout of the crossbar switch is shown in Figure 1.12. There are several examples of systems using crossbar switches at some level with the system, especially very high performance systems. One of our students built a very early crossbar switch multiple microprocessor system in the 1970s (Wilkinson and Abachi, 1983).

Tree Networks. Another switch configuration is to use a *binary tree*, as shown in Figure 1.13. Each switch in the tree has two links connecting to two switches below it as the network fans out from the root. This particular tree is a *complete binary tree* because every level is fully occupied. The *height* of a tree is the number of links from the root to the lowest leaves. A key aspect of the tree structure is that the height is logarithmic; there are $\log_2 p$ levels of switches with p processors (at the leaves). The tree network need not be complete or based upon the base two. In an m-ary tree, each node connects to m nodes beneath it.

Under uniform request patterns, the communication traffic in a tree interconnection network increases toward the root, which can be a bottleneck. In a *fat tree network* (Leiserson, 1985), the number of the links is progressively increased toward the root. In a *binary fat tree*, we simply add links in parallel, as required between levels of a binary tree, and increase the number of links toward the root. Leiserson developed this idea into the *universal fat tree*, in which the number of links between nodes grows exponentially toward the root, thereby allowing increased traffic toward the root and reducing the communication bottleneck. The most notable example of a computer designed with tree interconnection networks is the Thinking Machine's Connection Machine CM5 computer, which uses a 4-ary fat tree (Hwang, 1993). The fat tree has been used subsequently. For example, the Quadrics QsNet network (see http://www.quadrics.com) uses a fat tree.

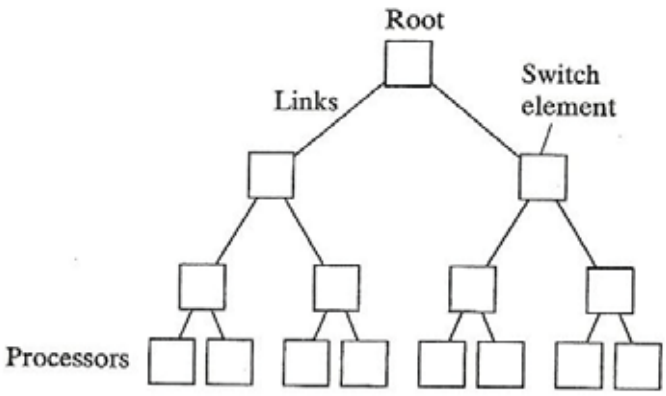


Figure 1.13 Tree structure.

Multistage Interconnection Networks. The multistage interconnection network (MIN) is a classification covering a multitude of configurations with the common characteristic of having a number of levels of switches. Switches in one level are connected to switches in adjacent levels in various symmetrical ways such that a path can made from one side of the network to the other side (and back sometimes). An example of a multistage interconnection network is the Omega network shown in Figure 1.14 (for eight inputs and outputs). This network has a very simple routing algorithm using the destination address. Inputs and outputs are given addresses as shown in the figure. Each switching cell requires one control signal to select either the upper output or the lower output (0 specifying the upper output and 1 specifying the lower). The most significant bit of the destination address is used to control the switch in the first stage; if the most significant bit is 0, the upper output is selected, and if it is 1, the lower output is selected. The next-most significant bit of the destination address is used to select the output of the switch in the next stage, and so on until the final output has been selected. The Omega network is highly blocking, though one path can always be made from any input to any output in a free network.

Multistage interconnection networks have a very long history and were originally developed for telephone exchanges, and are still sometimes used for interconnecting computers or groups of computers for really large systems. For example, the ASCI White supercomputer uses an Omega multistage interconnection network. For more information on multistage interconnection networks see Duato, Yalamanchili, and Ni (1997).

Communication Methods. The ideal situation in passing a message from a source node to a destination node occurs when there is a direct link between the source node and the destination node. In most systems and computations, it is often necessary to route a message through intermediate nodes from the source node to the destination node. There are two basic ways that messages can be transferred from a source to a destination: *circuit switching* and *packet switching*.

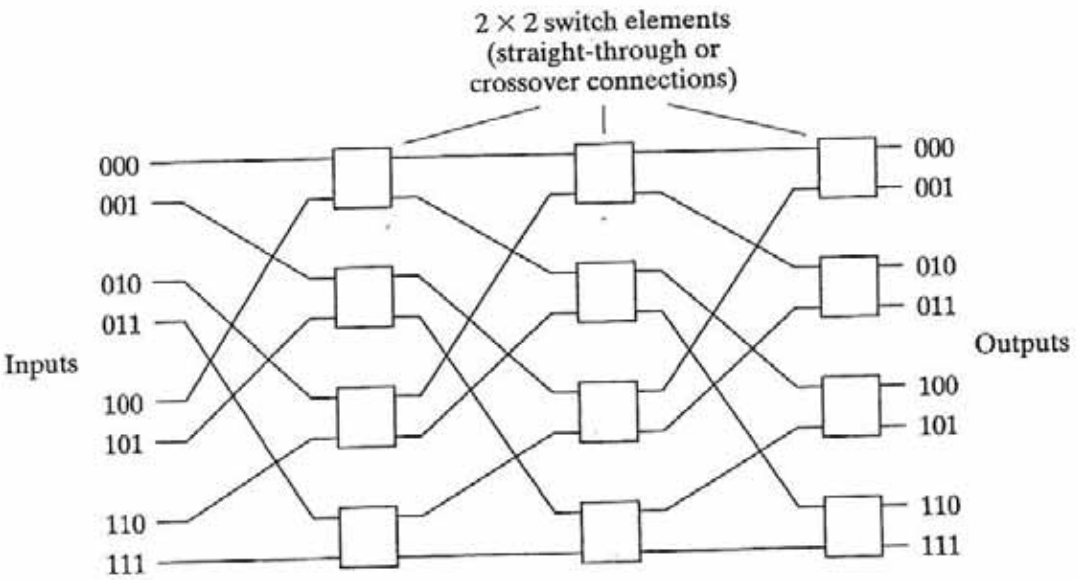


Figure 1.14 Omega network.

Circuit switching involves establishing the path and maintaining all the links in the path for the message to pass, uninterrupted, from the source to the destination. All the links are reserved for the transfer until the message transfer is complete. A simple telephone system (not using advanced digital techniques) is an example of a circuit-switched system. Once a telephone connection is made, the connection is maintained until the completion of the telephone call. Circuit switching has been used on some early multicomputers (e.g., the Intel IPSC-2 hypercube system), but it suffers from forcing all the links in the path to be reserved for the complete transfer. None of the links can be used for other messages until the transfer is completed.

In packet switching, the message is divided into "packets" of information, each of which includes the source and destination addresses for routing the packet through the interconnection network, and the data. There is a maximum size for the packet, say 1000 data bytes. If the message is larger than the maximum size, the message is broken up into separate packets, and each packet is sent through the network separately. Buffers are provided inside nodes to hold packets before they are transferred onward to the next node. A packet remains in a buffer if blocked from moving forward to the next node. The mail system is an example of a packet-switched system. Letters are moved from the mailbox to the post office and handled at intermediate sites before being delivered to the destination. This form of packet switching is called *store-and-forward packet switching* . Store-and-forward packet switching enables links to be used by other packets once the current packet has been forwarded. Unfortunately, store-and-forward packet switching, as described, incurs a significant latency, since packets must first be stored in buffers within each node, whether or not an outgoing link is available. This requirement is eliminated in *cut-through*, a technique originally developed for computer networks (Kermani and Kleinrock, 1979). In cut-through, if the outgoing link is available, the message is immediately passed forward without being stored in the nodal buffer; that is, it is "cut through." Thus, if the complete path were available, the message would pass immediately through to the destination. Note, however, that if the path is blocked, storage is needed for the complete message/packet being received.

Seitz introduced *wormhole* routing (Dally and Seitz, 1987) as an alternative to normal store-and-forward routing to reduce the size of the buffers and decrease the latency. In store-and-forward packet routing, a message is stored in a node and transmitted as a whole when an outgoing link becomes free. In wormhole routing, the message is divided into smaller units called *flits* (flow control digits). A flit is usually one or two bytes (Leighton, 1992). The link between nodes may provide for one wire for each bit in the flit so that the flit can be transmitted in parallel. Only the head of the message is initially transmitted from the source node to the next node when the connecting link is available. Subsequent flits of the message are transmitted when links become available, and the flits can be distributed through the network. When the head flit moves forward, the next one can move forward, and so on. A request/acknowledge signaling system is necessary between nodes to "pull" the flits along. When a flit is ready to move on from its buffer, it makes a request to the next node. When this node has a flit buffer empty, it calls for the flit from the sending node. It is necessary to reserve the complete path for the message as the parts of the message (the flits) are linked. Other packets cannot be interleaved with the flits along the same links.

Wormhole routing requires less storage at each node and produces a latency that is independent of the path length. Ni and McKinley (1993) present an analysis to show the independence of path length on latency in wormhole routing. If the length of a flit is much less than the total message, the latency of wormhole routing will be appropriately

constant irrespective of the length of the route. (Circuit switching will produce a similar characteristic.) In contrast, store-and-forward packet switching produces a latency that is approximately proportional to the length of the route, as is illustrated in Figure 1.15.

Interconnection networks, as we have seen, have routing algorithms to find a path between nodes. Some routing algorithms are adaptive in that they choose alternative paths through the network depending upon certain criteria, notably local traffic conditions. In general, routing algorithms, unless properly designed, can be prone to *livelock* and *deadlock*. *Livelock* can occur particularly in adaptive routing algorithms and describes the situation in which a packet keeps going around the network without ever finding its destination. *Deadlock* occurs when packets cannot be forwarded to the next node because they are blocked by other packets waiting to be forwarded, and these packets are blocked in a similar way so that none of the packets can move.

Deadlock can occur in both store-and-forward and wormhole networks. The problem of deadlock appears in communication networks using store-and-forward routing and has been studied extensively in that context. The mathematical conditions and solutions for deadlock-free routing in any network can be found in Dally and Seitz (1987). A general solution to deadlock is to provide *virtual* channels, each with separate buffers, for classes of messages. The *physical* links or channels are the actual hardware links between nodes. Multiple *virtual* channels are associated with a physical channel and time-multiplexed onto the physical channel, as shown in Figure 1.16. Dally and Seitz developed the use of separate virtual channels to avoid deadlock for wormhole networks.

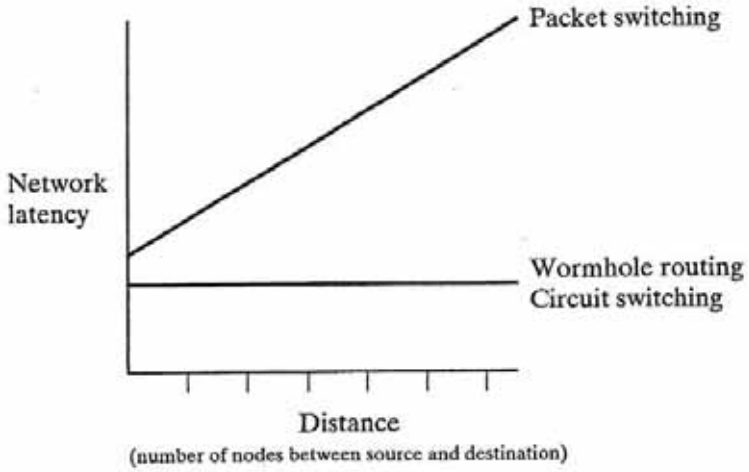


Figure 1.15 Network delay characteristics.

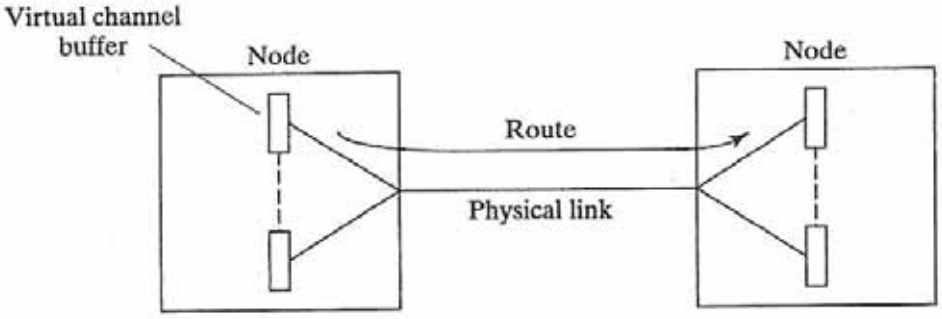


Figure 1.16 Multiple virtual channels mapped onto a single physical channel.

1.3.3 Distributed Shared Memory

The message-passing paradigm is often not as attractive for programmers as the shared memory paradigm. It usually requires the programmers to use explicit message-passing calls in their code, which is very error prone and makes programs difficult to debug. Message-passing programming has been compared to low-level assembly language programming (programming using the internal language of a processor). Data cannot be shared; it must be copied. This may be problematic in applications that require multiple operations across large amounts of data. However, the message-passing paradigm has the advantage that special synchronization mechanisms are not necessary for controlling simultaneous access to data. These synchronization mechanisms can significantly increase the execution time of a parallel program.

Recognizing that the shared memory paradigm is desirable from a programming point of view, several researchers have pursued the concept of a *distributed shared memory system.* As the name suggests, in a distributed shared memory system the memory is physically distributed with each processor, but each processor has access to the whole memory using a single memory address space. For a processor to access a location not in its local memory, message passing occurs to pass data between the processor and the memory location but in some automated way that hides the fact that the memory is distributed. Of course, accesses to remote locations will incur a greater delay, and usually a significantly greater delay, than for local accesses.

Multiprocessor systems can be designed in which the memory is physically distributed but operates as shared memory and appears from the programmer's perspective as shared memory. A number of projects have been undertaken to achieve this goal using specially designed hardware, and there have been commercial systems based upon this idea. Perhaps the most appealing approach is to use networked computers. One way to achieve distributed shared memory on a group of networked computers is to use the existing virtual memory management system of the individual computers which is already provided on almost all systems to manage its local memory hierarchy. The virtual memory management system can be extended to gives the illusion of global shared memory even when it is distributed in different computers.This idea is called *shared virtual memory.* One of the first to develop shared virtual memory was Li (1986). There are other ways to achieve distributed shared memory that do not require the use of the virtual memory management system or special hardware. In any event, physically the system is as given for message-passing multicomputers in Figure 1.8, except that now the local memory becomes part of the shared memory and is accessible from all processors, as illustrated in Figure 1.17.

Implementing and programming a distributed shared memory system is considered in detail in Chapter 9 after the fundamental concepts of shared memory programming in Chapter 8. Shared memory and message passing should be viewed as programming paradigms in that either could be the programming model for any type of multiprocessor, although specific systems may be designed for one or the other.

It should be mentioned that DSM implemented on top of a message-passing system usually will not have the performance of a true shared memory system, nor will using message-passing directly on a message system.

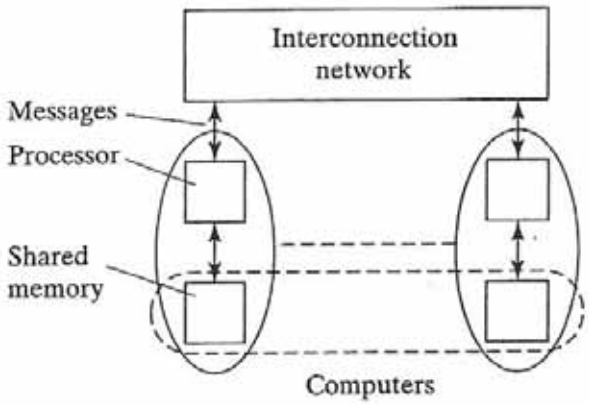


Figure 1.17 Shared memory multiprocessor.

1.3.4 MIMD and SIMD Classifications

In a single-processor computer, a single stream of instructions is generated by the program execution. The instructions operate upon data items. Flynn (1966) created a classification for computers and called this single-processor computer a *single instruction stream-single data stream* (SISD) computer. In a general-purpose multiprocessor system, each processor has a separate program, and one instruction stream is generated from each program for each processor. Each instruction operates upon different data. Flynn classified this type of computer as a *multiple instruction stream-multiple data stream* (MIMD) computer. The shared memory and message-passing multiprocessors so far described are both in the MIMD classification. The term MIMD has stood the test of time and is still widely used for a computer system operating in this mode.

Apart from the two extremes, SISD and MIMD, for certain applications there can be significant performance advantages in designing a computer in which a single instruction stream is from a single program but multiple data streams exist. The instructions from the program are broadcast to more than one processor. Each processor is essentially an arithmetic processor without a (program) control unit. A single control unit is responsible for fetching the instructions from memory and issuing the instructions to the processors. Each processor executes the same instruction in synchronism, but using different data. For flexibility, individual processors can be inhibited from participating in the instruction. The data items form an array, and an instruction acts upon the complete array in one instruction cycle. Flynn classified this type of computer as a *single instruction stream-multiple data stream* (SIMD) computer. The SIMD type of computer was developed because there are a number of important applications that mostly operate upon arrays of data. For example, most computer simulations of physical systems (from molecular systems to weather forecasting) start with large arrays of data points that must be manipulated. Another important application area is low-level image processing, in which the picture elements (pixels) of the image are stored and manipulated, as described in Chapter 12. Having a system that will perform similar operations on data points at the same time will be both efficient in hardware and relatively simple to program. The program simply consists of a single sequence of instructions operating on the array of data points together with normal control instructions executed by the separate control unit. We will not consider SIMD computers in this text as they are specially designed computers, often for specific applications. Computers today can have SIMD instructions for multimedia and graphics applications. For example, the

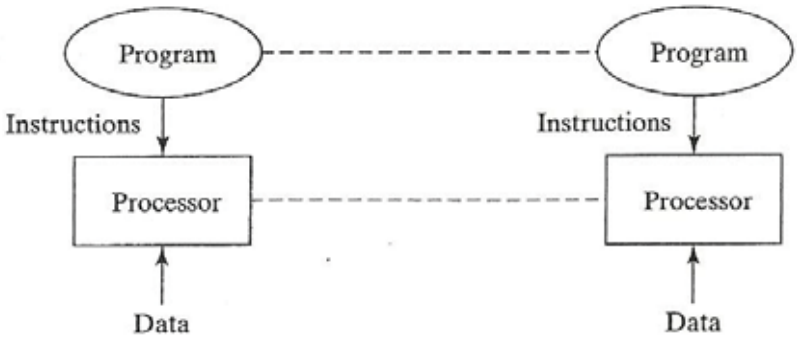


Figure 1.18 MPMD structure.

Pentium family, starting with the Pentium II, now has such SIMD instructions added to speed up multimedia and other applications that require the same operation to be performed on different data, the so-called MMX (MultiMedia eXtension) instructions.

The fourth combination of Flynn's classification, *multiple instruction stream-single data stream* (MISD) computer, does not exist unless one specifically classifies pipelined architectures in this group, or possibly some fault tolerant systems.

Within the MIMD classification, which we are concerned with, each processor has its own program to execute. This could be described as *multiple program multiple data* (MPMD) structure, as illustrated in Figure 1.18. Of course, all of the programs to be executed could be different, but typically only two source programs are written, one for a designated master processor and one for the remaining processors, which are called slave processors. A programming structure we may use, or may have to use, is the *single program multiple data* (SPMD) structure. In this structure, a single source program is written and each processor will execute its personal copy of this program, although independently and not in synchronism. The source program can be constructed so that parts of the program are executed by certain computers and not others depending upon the identity of the computer. For a master-slave structure, the program would have parts for the master and parts for the slaves.

1.4 CLUSTER COMPUTING

1.4.1 Interconnected Computers as a Computing Platform

So far, we have described specially designed parallel computers containing multiple processors or multiple computers as the computing platform for parallel computing. There have been numerous university research projects over the years designing such multiprocessor systems, often with radically different architectural arrangements and different software solutions, each project searching for the best performance. For large systems, the direct links have been replaced with switches and multiple levels of switches (multistage interconnection networks). Computer system manufacturers have come up with numerous designs. The major problem that most manufacturers have faced is the unending progress towards faster and faster processors. Each new generation of processors is faster and able to perform more simultaneous operations internally to boost performance. The most

obvious improvement noticed by the computer purchaser is the increase in the clock rate of personal computers. The basic clock rate continues to increase unabated. Imagine purchasing a Pentium (or any other) computer one year and a year later being able to purchase the same system but with twice the clock frequency. And in addition to clock rate, other factors make the system operate even faster. For example, newer designs may employ more internal parallelism within the processor and other ways to achieve faster operation. They often use memory configurations with higher bandwidth. The way around the problem of unending progress of faster processors for "supercomputer" manufacturers has been to use a huge number of available processors. For example, suppose a multiprocessor is designed with state-of-the-art processors in 2004, say 3GHz processors. Using 500 of these processors together should still overtake the performance of any single processor system for some years, but at an enormous cost.

In the late 1980s and early 1990s, another more cost-effective approach was tried by some universities—using workstations and personal computers connected together to form a powerful computing platform. A number of projects explored forming groups of computers from various perspectives. Some early projects explored using workstations as found in laboratories to form a *cluster of workstations* (COWs) or *network of workstations* (NOWs), such as the NOW project at Berkeley (Anderson, Culler, and Patterson, 1995). Some explored using the free time of existing workstations when they were not being used for other purposes, as oftentimes workstations, especially those in offices, are not used continuously or do not require 100% of the processor time even when they are being used.

Initially, using a network of workstations for parallel computing became interesting to many people because networks of workstations already existed for general-purpose computing. Workstations, as the name suggests, were already used for various programming and computer-related activities. It was quickly recognized that a network of workstations, offered a very attractive alternative to expensive supercomputers and parallel computer systems for high-performance computing. Using a network of workstations has a number of significant and well-enumerated advantages over specially designed multiprocessor systems. Key advantages are:

1. Very high performance workstations and PCs are readily available at low cost.
2. The latest processors can easily be incorporated into the system as they become available and the system can be expanded incrementally by adding additional computers, disks, and other resources.
3. Existing application software can be used or modified.

Software was needed to be able to use the workstations collectively, and fortuitously, at around the same time, message-passing tools were developed to make the concept usable. The most important message-passing project to provide parallel programming software tools for these workstations was Parallel Virtual Machine (PVM), started in the late 1980s. PVM was a key enabling technology and led to the success of using networks of workstations for parallel programming. Subsequently, the standard message-passing library, Message-Passing Interface (MPI), was defined.

The concept of using multiple interconnected personal computers (PCs) as a parallel computing platform matured in the 1990s as PCs became very inexpensive and powerful. Workstations, that is, computers particularly targeted towards laboratories, were being

replaced in part by regular PCs, and the distinction between workstations and PCs in general-purpose laboratories disappeared. The term "network of workstations" has given way to simply a "cluster" of computers, and using the computers in a cluster collectively on a single problem by the term *cluster computing*.[3]

Ethernet Connections. The communication method for networked computers has commonly been an Ethernet type, which originally consisted of a single wire to which all the computers attach, as shown in Figure 1.19. Shown here is a file server that holds all the files of the users and the system utilities. The use of a single wire was regarded as a cost and layout advantage of the Ethernet design. Nowadays, a single wire has been replaced with various switches and hubs while maintaining the Ethernet protocol. A switch, as the name suggests, provides direct switched connections between the computers to allow multiple simultaneous connections, as illustrated in Figure 1.20, whereas a hub is simply a point where all the computers are connected. The switch automatically routes the packets to their destinations and allows multiple simultaneous connections between separate pairs of computers. Switches are interconnected in various configurations to route messages between the computers in the network.

In the Ethernet type of connection, all transfers between a source and a destination are carried in packets serially (one bit after another on one wire). The packet carries the source address, the destination address, and the data. The basic Ethernet format is shown in

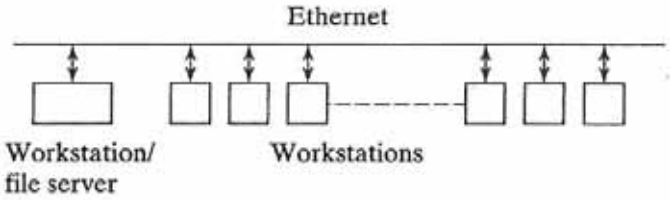


Figure 1.19 Original Ethernet-type single wire network.

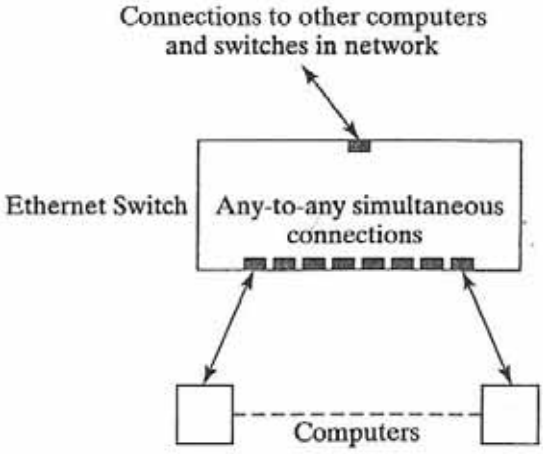


Figure 1.20 Ethernet switch.

[3] Although the term "cluster computing" is now the accepted term, it has been applied to networks of workstations/PCs being used collectively to solve problems since the early 1990s. For example, there were workshops called Cluster Computing at the Supercomputing Computations Research Institute at Florida State University in 1992 and 1993.

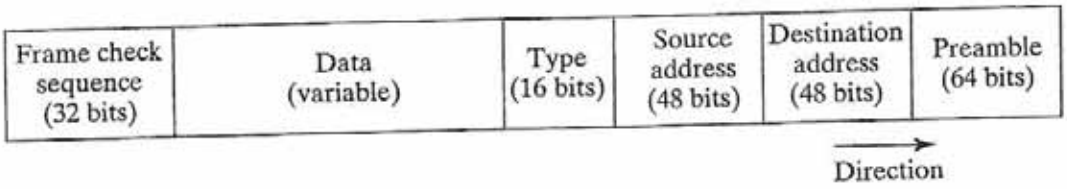


Figure 1.21 Ethernet frame format.

Figure 1.21. The preamble shown in Figure 1.21 is for synchronization. There is a maximum size for the data (1.5 K bytes), and if the data to be sent is larger than that, it is divided in separate packets, each with its source and destination address.[4] Packets could take different paths from the source to the destination and would often do so on a large network or the Internet and have to be reconstituted in the correct order at the destination.

As mentioned, the original Ethernet protocol was designed to use a single wire connecting multiple computers. Since each workstation is operating completely independently and could send messages at any time, the Ethernet line may be needed by one computer while it is already being used to carry packets sent by other another computer. Packets are not sent if it can be detected that information is already being transmitted on the network. It may be that at the time a packet is to be sent, no other information is passing along the Ethernet line at the point where this computer is attached and hence it will launch its packet. However, more than one packet could be launched by different workstations at nearly the same instant. If more than one packet is submitted to the network, the information from them will be corrupted. This is detected at the source by simply comparing the information being sent to that actually on the Ethernet line. If not the same, the individual packets are resubmitted after intervals, all according to the Ethernet protocol (IEEE standard 802.3).

The original speed for Ethernet was 10 Mbits/sec, which has been improved to 100 Mbits/sec and 1000 Mbits/sec (the latter called Gigabit Ethernet). The interconnects can be twisted-pair wire (copper), coax wire, or optical fiber, the latter for higher speed and longer distances. We should mention that the message latency is very significant with Ethernet, especially with the additional overhead caused by some message-passing software.

Network Addressing. TCP/IP (Transmission Control Protocol/Internet Protocol) is a standard that establishes the rules for networked computers to communicate and pass data. On the Internet, each "host" computer is given an address for identification purposes. TCP/IP defines the format of addresses as a 32-bit number divided into four 8-bit numbers (for IPv4, Internet Protocol version 4). With certain constraints, each number is in the range 0–255. The notation for the complete address is to separate each number by a period. For example, a computer might be given the IP address:

```
129.49.82.1
```

In binary, this address would be:

```
10000001.00110001.01010010.00000001
```

[4] It is possible to increase the packet size. Alteron Networks has a propriety technique called jumbo frames to increase the packet size from 1,500 bytes to 9,000 bytes.

The address is divided into fields to select a network, a possible sub-network, and computer ("host") within the sub-network or network. There are several formats identified by the first one, two, three, or four bits of the address. The layout of the IPv4 formats are shown in Figure 1.22. This information is relevant for setting up a cluster.

Class A format is identified with a leading 0 in the address and uses the next seven bits as the network identification. The remaining 24 bits identify the sub-network and "host" (computer). This provides for 16,777,216 (2^{24}) hosts within a network, but only 128 networks are available. The hosts can be arranged in various sub-network configurations. Class A would be used for very large networks.

Class B is for medium-sized networks and identified by the first two bits being 10. The network is identified by the next 14 bits. The remaining 16 bits are used for the sub-network and host. This provides for 65,536 (2^{16}) hosts within a network, and 16,384 (2^{14}) networks are available. Again the hosts can be arranged in various sub-network configurations, but a simple configuration would be to have 256 sub-networks and 256 hosts in each sub-network; that is, the first eight-bits of the sub-network/host part to identify the sub-network and the remaining eight bits to identify the host within the sub-network.

Class C is for small networks and identified by the first three bits being 110. The network is identified by the next 21 bits. The remaining eight bits are used for the host. This provides for 256 (2^8) hosts within a network, and 2,097,152 (2^{21}) networks are available.

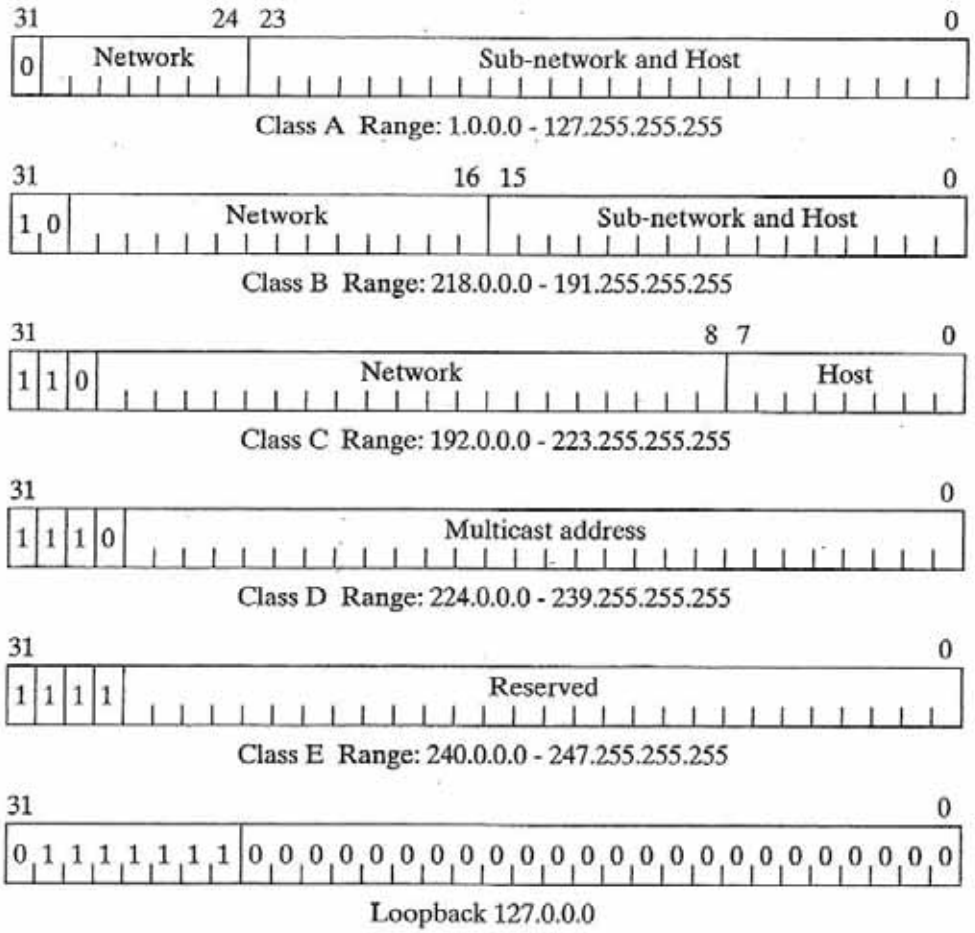


Figure 1.22 IPv4 formats.

The hosts can be arranged in various sub-network configurations, but a simple configuration would not to have a sub-network.

Class D is used to broadcast a message to multiple destinations simultaneously; that is, the transmission is picked up by multiple computers (called multicast). The loopback format is used to send a message back to oneself for testing. Certain addresses are reserved, as indicated in Figure 1.22, and some network addresses within classes A, B, and C are reserved for private networks (10.0.0.0 to 10.255.255.255, 172.16.0.0 to 172.32.255.255, and 192.168.0.0 to 192.168.255.255). Private network addresses can be used on dedicated clusters, as will be discussed later.

IPv4 with its 32-bit addresses provides for about 4 billion hosts (2^{32} = 4,294,967,296, less those not used for specific host addresses). The Internet has grown tremendously, to over 100,000,000 hosts by 2001 by most estimates (Knuckles, 2001), and soon more IP addresses will be needed. Not only are IP addresses used for computers connected to the Internet permanently, as in computer laboratories; IP addresses are also used by Internet Service Providers for dial-up and other connections to customers. IPv6 (Internet Protocol version 6) has been developed to extend the addressability of IPv4 by using 128 bits divided into eight 16-bit sections. This gives 2^{128} possible hosts (a big number!). IPv6 also has a number of other enhancements for message transfers. Network software can be designed to handle both IPv4 and IPv6. For the following, we will assume IPv4 addresses.

The IP addressing information is important to setting up a cluster because IP addressing is usually used to communicate between computers within the cluster and between the cluster and users outside the cluster. Network addresses are assigned to the organization by the Internet Assigned Number Authority. The sub-network and host assignments are chosen by the organization (i.e., its system administrator for the sub-network/host). Masks are set up in the communication software to select the network, sub-network, and host field. The masks are 32-bit numbers with 1's defining the network/sub-network part of the address. For example, the mask for a class B address with bits 8 to 15 (in Figure 1.22) used for the sub-network would be:

```
255.255.255.0
```

or in binary:

```
11111111.11111111.11111111.00000000
```

which is used to separate the host address from the network/sub-network address. Note that the division of sub-network and host field need not be on 8-bit boundaries and is decided by the local system administrator, but the network address (A, B, or C) is allocated to the organization.

Computers connect to an Ethernet cable via a Ethernet network interface card (NIC). The source and destination addresses in the Ethernet format shown in Figure 1.21 are not IP addresses; they are the addresses of network interface cards. These addresses are 48 bits and called MAC (Media Access Controller) addresses. Each network interface card has a predefined and unique 48-bit MAC address that is set up during manufacture of the chip or card. (Allocation of addresses is controlled by the IEEE Registration Authority.) While the IP address of a computer is selected by software, the MAC address of each NIC is unalterable. A translation is necessary between the two to establish a communication path. The higher-level software will use IP addresses, and the lower-level network interface software

will use MAC addresses. Actually, both MAC and IP address are contained in the Ethernet packet, the IP addresses are within the data part of the packet in Figure 1.21.

There is a level above IP addressing whereby IP addresses are converted into names for ease of user interaction. For example, `sol.cs.wcu.edu`, is the name of one of Western Carolina University's servers within the Department of Mathematics and Computer Science; its IP address is `152.30.5.10`. The relationship between name and IP address is established using the Domain Naming Service, a distributed name database.[5]

1.4.2 Cluster Configurations

There are several ways a cluster can be formed.

Existing Networked Computers. One of the first ways to form a cluster was to use existing networked workstations in a laboratory, as illustrated in Figure 1.23(a). These workstations were already provided with IP addresses for network communication. Messaging software provided the means of communication. Indeed, the first way tried by the authors for teaching cluster computing in the early 1990s was to use existing networked computers. Using a network of existing computers is very attractive for educational institutions because it can be done without additional resources. However, it can present significant problems in the usage of the computers. Cluster computing involves using multiple computers simultaneously. Clearly, it is possible with modern operating systems to arrange for the computers to run the cluster computing programs in the background while other

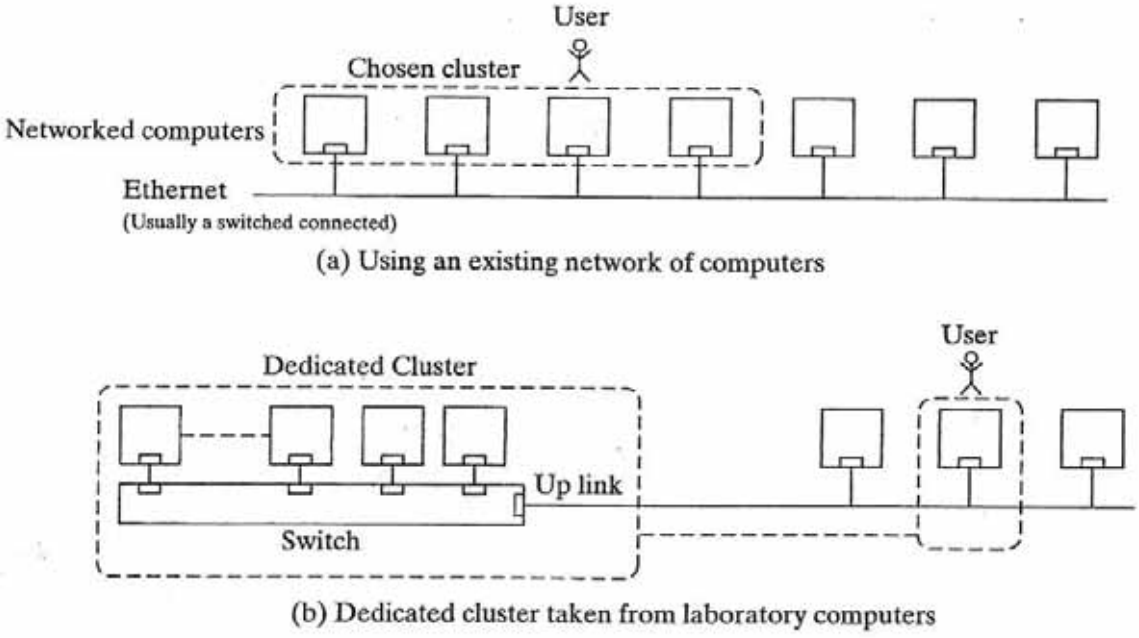


Figure 1.23 Early ways to form a cluster.

[5] In UNIX systems, the relationship between host name and IP address is held in a file called hosts, which can be inspected (e.g., cat /etc/hosts). A look-up table is maintained holding the relationship between the name/IP address and Ethernet MAC address of hosts. This table can be inspected with the address resolution protocol command arp -a.

users are directly working at the computer. Moreover, the structure of the message-passing software then used (PVM) made this easy. In practice, situations arise that make it unworkable. Users at the computer can cause the computer to stop while the cluster computing work is in progress (they can simply turn the computer off!). Conversely, cluster computing activities by students can cause the computers to get into difficulties. It also requires the ability for remote access to the computers, with possible security issues if not done properly. At the time, the common way for remote access (in UNIX) was through "r" commands (`rlogin`, `rsh`) which were used by the message-passing software to start processes remotely. Since these commands are insecure, students would be able to remotely access other computers and cause havoc. (Passwords were transmitted unencrypted.) More recently, of course, remote access has been made secure with the use of `ssh`.

Moving to a Dedicated Computer Cluster. We quickly found it very cost-effective (free!) and less trouble simply to move computers from a laboratory into a dedicated cluster when the computers were upgraded in the laboratory. Every time the laboratory was upgraded, so was the cluster, but with last year's models that were being replaced in the laboratory. The computers forming a cluster need not have displays or keyboards and are linked with the same communication medium as used in the laboratories. Simply moving computers into a dedicated group could be done without any changes to IP addresses. The computers could still belong to the sub-network as before except that each computer would never have local users sitting at its console. All access is done remotely. A user would login to a computer outside the cluster group and enroll the cluster computers together with its own computer to form a cluster, as illustrated in Figure 1.23(b). Note that the computers in the cluster would be the type that were originally selected for the computer laboratory. For example, our cluster formed that way originally consisted of eight SUN IPC computers in the early 1990s, which were upgraded to eight SUN Ultra computers later when these computers were being replaced in the general-purpose laboratories.

Beowulf Clusters. A small but very influential cluster-computing project was started at the NASA Goddard Space Flight Center in 1993, concentrating upon forming a cost-effective computer cluster by the use of readily available low-cost components. Standard off-the-shelf microprocessors were chosen with a readily available operating system (Linux) and connected together with low-cost interconnects (Ethernet). Whereas other projects were also concerned with constructing clusters, they often used some specialized components and software in their design to obtain the best performance. In contrast, the NASA project started with the premise that only widely available low-cost components should be used, chosen on a cost/performance basis. It was entitled the Beowulf project (Sterling, 2002a and 2002b). This name has stuck for describing any cluster of low-cost computers using commodity interconnects and readily available software for the purpose of obtaining a cost-effective computing platform. Originally, Intel processors (486's) were used and the free Linux operating system, and Linux is a still common operating system for Beowulf clusters with Intel processors. Other types of processors can be employed in a Beowulf cluster.

The key attribute for attaching the name Beowulf to a cluster is the use of widely available components to obtain the best cost/performance ratio. The term *commodity computer* is used to highlight the fact that the cost of personal computers is now so low that

computers can be bought and replaced at frequent intervals. The mass market for personal computers has made their manufacture much less expensive. And this applies to all the components around the processor, such as memory and network interfaces. We now have commodity Ethernet network interfaces cards (NICs) at minimal cost. Such interconnects can be used to connect the commodity computers to form a cluster of commodity computers.

Beyond Beowulf. Clearly, one would use higher-performance components if that made economic sense, and really high-performance clusters would use the highest-performance components available.

Interconnects. Beowulf clusters commonly use fast Ethernet in low-cost clusters. Gigabit Ethernet is an easy upgrade path; this was the choice made at UNC-Charlotte. However, there are more specialized and higher-performance interconnects, such as Myrinet, a 2.4 Gbits/sec interconnect. There are other interconnects that could be used, including cLan, SCI (Scalable Coherent Interface), QsNet, and Infiniband; see Sterling (2002a) for more details.

Clusters with Multiple Interconnects. The Beowulf project and other projects explored using multiple parallel interconnections to reduce the communication overhead. Clusters can be set up with multiple Ethernet cards or network cards of different types. The original Beowulf project used two regular Ethernet connections per computer and a "channel bonding" technique. Channel bonding associates several physical interfaces with a single virtual channel. Software is available to achieve this effect (e.g., see http://cesdis.gsfc.nasa.gov/beowulf/software). In the context of Beowulf, the resulting structure had to be cost-effective. It did show significant improvement in performance (see Sterling, 2002 for more details). Some recent clusters have used slower Ethernet connections for set-up and faster interconnects such as Myrinet for communication during program execution.

We have worked on the concept of using multiple Ethernet lines configured as shown in Figure 1.24(a), (b), and (c). There are numerous ways that switches can be used. The configurations shown are in the general classification of overlapping connectivity networks (Hoganson, Wilkinson, and Carlisle, 1997; Wilkinson, 1990, 1991, 1992a, 1992b). Overlapping connectivity networks have the characteristic that regions of connectivity are provided and the regions overlap. In the case of overlapping connectivity Ethernets, this is achieved by having Ethernet segments such as shown in the figure, but there are several other ways overlapping connectivity can be achieved; see, for example, Wilkinson and Farmer (1994). It should be mentioned that the structures of Figure 1.24 significantly reduce collisions but the latency and data transmission times remain.

Symmetrical Multiprocessors (SMP) Cluster. Small shared memory multiprocessors based around a bus, as described in Section 1.2.1, have a symmetry between their processors and memory modules and are called symmetric or symmetrical (shared memory) multiprocessors. Small shared memory multiprocessor systems based upon Pentium processors are very cost-effective, especially two-processor systems. Hence, it is also reasonable to form clusters of "symmetrical multiprocessor" (SMP) systems, as illustrated in Figure 1.25. This leads to some interesting possibilities for programming such a cluster. Between SMPs, message passing could be used, and within the SMPs, threads or other

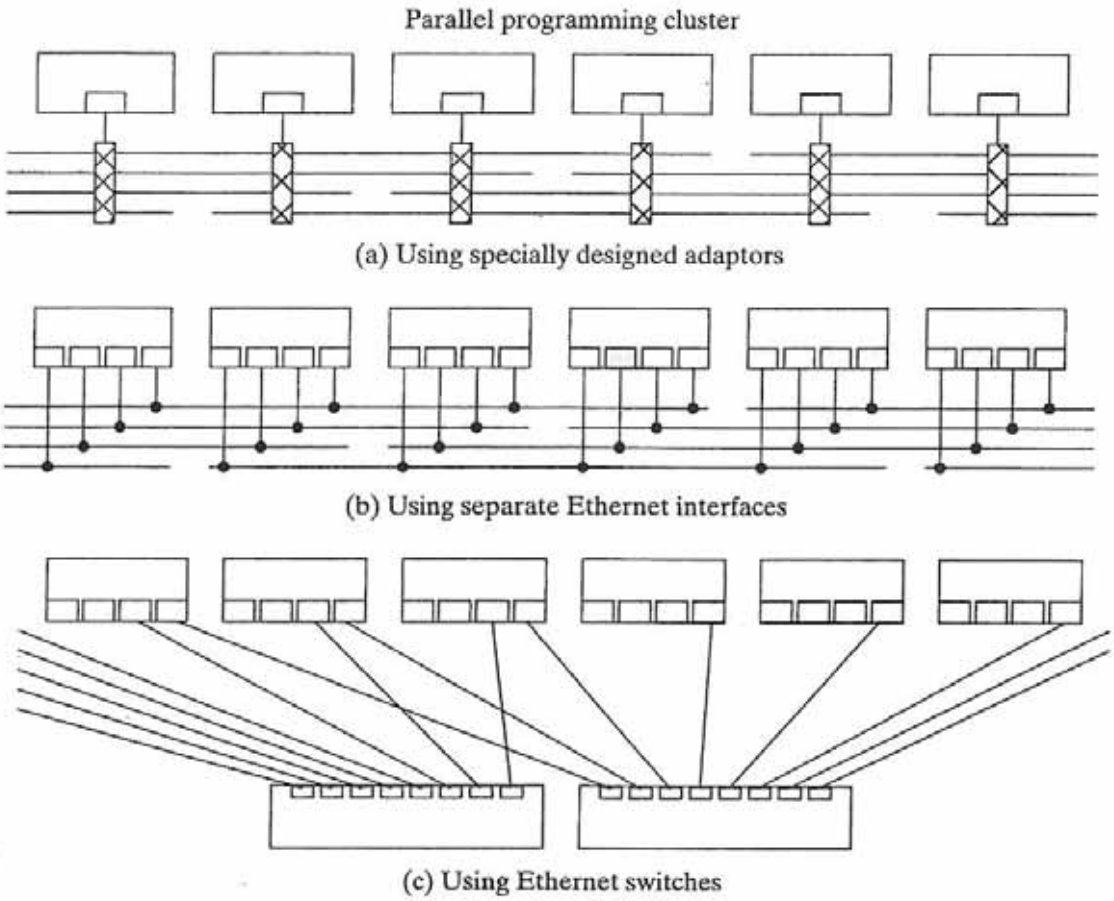


Figure 1.24 Overlapping connectivity Ethernets.

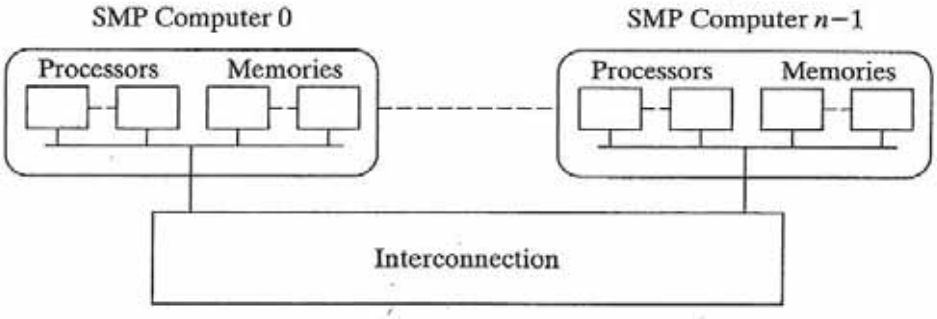


Figure 1.25 Cluster of shared memory computers.

shared memory methods could be used. Often, however, for convenience, message passing is done uniformly. When a message is to pass between processors within a SMP computer, the implementation might use shared memory locations to hold the messages, and communication would be much faster.

Web Clusters. Since the arrival of the Internet and the World Wide Web, computers in different locations and even countries are interconnected. The emergence of the Web has led to the possibility of using computers attached to the Web at various sites for parallel programming. Projects have investigated using the "web" of computers to form a

parallel computing platform. The idea was originally called *metacomputing* and is now called *grid computing*. Projects involved in this type of large-scale cluster computing include Globus, Legion, and WebFlow. More details of these three systems can be found in Baker and Fox (1999).

1.4.3 Setting Up a Dedicated "Beowulf Style" Cluster

"Beowulf style" implies commodity components. These are generally PCs that can be bought from well-known suppliers. These suppliers have now embraced cluster computing and offer pre-packaged cluster computing systems, although they may be targeted towards very high performance using multiple dual/quad processor servers. In any event, the setup procedures have been substantially simplified with the introduction of software packages such as Oscar, which automates the procedure of loading the operating system and other procedures. We shall briefly outline Oscar later in this section.

Hardware Configuration. A common hardware configuration is to have one computer operating as a master node with the other computers in the cluster operating as compute nodes within a private network. The master node is traditionally known as the *frontend* and acts as a file server with significant memory and storage capacity. Generally, it is convenient for all the compute nodes to have disk drives, although diskless compute nodes are possible. Connection between the compute nodes and the master node can be by Fast or Gigabit Ethernet (or possibly a more specialized interconnect, such as Myrinet), as illustrated in Figure 1.26. The master node needs a second Ethernet interface to connect to the outside world and would have a globally accessible IP address. The compute nodes would be given private IP addresses; that is, these computers can only communicate within the cluster network and are not directly accessible from outside the cluster.

This model can be enhanced in several ways. Another computer acting as an administrative or management node can be added. This node would be used by the system administrator for monitoring the cluster and testing. The sole purpose of the compute nodes is to perform computations, so they do not need a keyboard or display. However, it may be convenient to be able to access each compute node through its normal console input. Hence, the serial connections of these nodes could be brought back to the master though a serial concentrator switch or, if present, the administrative node. There are various possible connections. Figure 1.27 shows one arrangement whereby a single display and keyboard are present that can be switched between the master node and the administrative node.

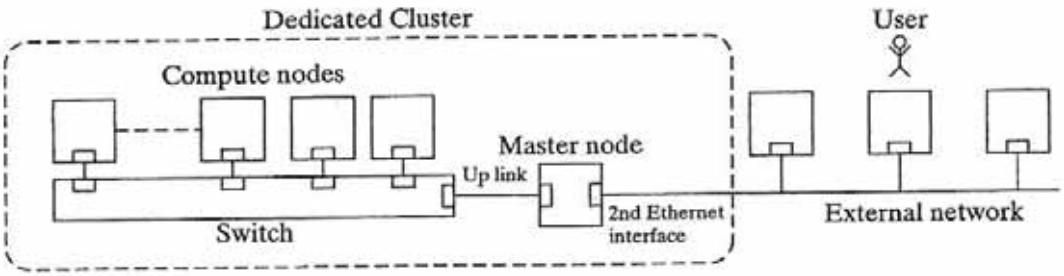


Figure 1.26 Dedicated cluster with a master node.

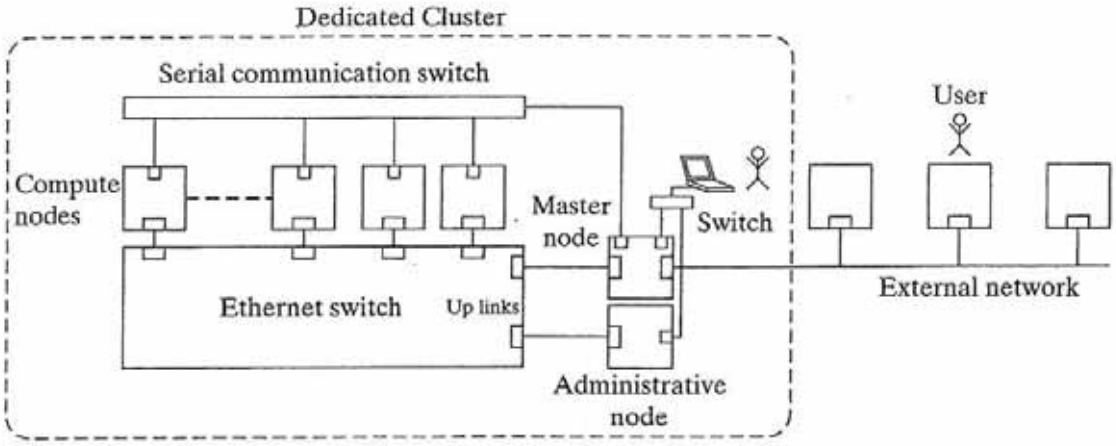


Figure 1.27 Dedicated cluster with a master and administrative nodes and serial connections.

Software Configuration. Normally every computer in the cluster will have a copy of the operating system (traditionally Linux, but Windows clusters can also be formed). The master node will normally also contain all the application files for the cluster and be configured as a file server using a network file system, which allows the compute nodes to see and have direct access to files stored remotely. The most commonly used network file system is NFS. Note that the cluster compute nodes are separated from the outside network and all user access is through the master node via a separate Ethernet interface and IP address. Mounted on the master node will be the message-passing software (MPI and PVM, which will be discussed in Chapter 2), cluster-management tools, and parallel applications. The message-passing software is between the operating system and the user, and is the *middleware*.

Once all the software is established, the user can log onto the master node and enroll compute nodes using the message-passing software, again as will be described in Chapter 2. The challenge here is to set up the cluster in the first place, which involves fairly detailed operating system and networking knowledge (i.e., Linux commands and how to use them). There are books and Web sites dedicated to this task (and even workshops). Fortunately, as mentioned, the task of setting up the software for a cluster has been substantially simplified with the introduction of cluster setup packages, including Oscar (Open Source Cluster Application Resources), which is menu driven and freely available. Before starting with Oscar, the operating system (RedHat Linux) has to be mounted on the master mode. Then, in a number of menu-driven steps, Oscar mounts the required software and configures the cluster. Briefly, NFS and network protocols are set up on the master node. The cluster is defined in a database. The private cluster network is defined with IP addresses selected by the user. The Ethernet interface MAC addresses of the compute nodes are collected. They are obtained by each compute node making a Boot Protocol (BOOTP/DHCP) request to the master node. IP addresses are returned for the compute nodes. Also returned is the name of the "boot" file specifying which kernel (the central part of operating system) to boot. The kernel is then downloaded and booted, and the compute node file system created. The operating system on the compute nodes is installed through the network using the Linux installation utility LUI. Finally, the cluster is configured and middleware installed

and configured. Test programs are run. The cluster is then ready. Workload management tools are provided for batch queues, scheduling, and job monitoring. Such tools are very desirable for a cluster. More details of Oscar can be found at http://www.csm.ornl.gov/oscar.

1.5 SUMMARY

This chapter introduced the following concepts:

- Parallel computers and programming
- Speedup and other factors
- Extension of a single processor system into a shared memory multiprocessor
- The message-passing multiprocessor (multicomputer)
- Interconnection networks suitable for message-passing multicomputers
- Networked workstations as a parallel programming platform
- Cluster computing

FURTHER READING

Further information on the internal design of multiprocessor systems can be found in computer architecture texts such as Culler and Singh (1999), Hennessy and Patterson (2003), and Wilkinson (1996). A great deal has been published on interconnection networks. Further information on interconnection networks can be found in a significant textbook by Duato, Yalamanchili, and Ni (1997) devoted solely to interconnection networks. An early reference to the Ethernet is Metcalfe and Boggs (1976).

Anderson, Culler, and Patterson (1995) make a case for using a network of workstations collectively as a multiple computer system. Web-based material on workstation cluster projects includes http://cesdis.gsfc.nasa.gov/beowulf. An example of using shared memory on networked workstations can be found in Amza et al. (1996).

Recognizing the performance limitation of using commodity interfaces in workstation clusters has led several researchers to design higher-performance network interface cards (NICs). Examples of work in this area includes Blumrich et al. (1995), Boden et al. (1995), Gillett and Kaufmann (1997), and Minnich, Burns, and Hady (1995). Martin et al. (1997) have also made a detailed study of the effects of communication latency, overhead, and bandwidth in clustered architecture. One point they make is that it may be better to improve the communication performance of the communication system rather than invest in doubling the machine performance.

The two-volume set edited by Buyya (1999a and 1999b) provides a wealth of information on clusters. Williams (2001) wrote an excellent text on computer system architecture with an emphasis on networking. Details about building a cluster can be found in Sterling (2002a and 2002b).

BIBLIOGRAPHY

AMDAHL, G. (1967), "Validity of the Single-Processor Approach to Achieving Large-Scale Computing Capabilities," *Proc. 1967 AFIPS Conf.*, Vol. 30, p. 483.

AMZA, C., A. L. COX, S. DWARKADAS, P. KELEHER, H. LU, R. RAJAMONY, W. YU, AND W. ZWAENEPOEL (1996), "TreadMarks: Shared Memory Computing on Networks of Workstations," *Computer*, Vol. 29, No. 2, pp. 18–28.

ANDERSON, T. E., D. E. CULLER, AND D. PATTERSON (1995), "A Case for NOW (Networks of Workstations," *IEEE Micro*, Vol. 15, No. 1, pp. 54–64.

BLUMRICH, M. A., C. DUBNICKI, E. W. FELTON, K. LI, AND M. R. MESRINA (1995), "Virtual-Memory-Mapped Network Interface," *IEEE Micro*, Vol. 15, No. 1, pp. 21–28.

BAKER, M., AND G. FOX (1999), "Chapter 7 Metacomputing: Harnessing Informal Supercomputers," *High Performance Cluster Computing*, Vol. 1 *Architecture and Systems*, (Editor BUYYA, R.), Prentice Hall PTR, Upper Saddle River, NJ.

BODEN, N. J., D. COHEN, R. E. FELDERMAN, A. E. KULAWIK, C. L. SEITZ, J. N. SEIZOVIC, AND W.-K. SU (1995), "Myrinet: A Gigabit-per-Second Local Area Network," *IEEE Micro*, Vol. 15, No. 1, pp. 29–36.

BUYYA, R., Editor (1999a), *High Performance Cluster Computing*, Vol. 1 *Architecture and Systems*, Prentice Hall PTR, Upper Saddle River, NJ.

BUYYA, R. Editor (1999b), *High Performance Cluster Computing*, Vol. 2 *Programming and Applications*, Prentice Hall PTR, Upper Saddle River, NJ.

CHANDRA, R., L. DAGUM, D. KOHR. D. MAYDAN, J. MCDONALD, AND R. MENON (2001), *Parallel Programming in OpenMP*, Morgan Kaufmann Publishers, San Francisco, CA.

CONWAY, M. E. (1963), "A Multiprocessor System Design," *Proc. AFIPS Fall Joint Computer Conf.*, Vol. 4, pp. 139–146.

CULLER, D. E., AND J. P. SINGH (1999), *Parallel Computer Architecture A Hardware/Software Approach*, Morgan Kaufmann Publishers, San Francisco, CA.

DALLY, W., AND C. L. SEITZ (1987), "Deadlock-free Message Routing in Multiprocessor Interconnection Networks," *IEEE Trans. Comput.*, Vol. C-36, No. 5, pp. 547–553.

DUATO, J., S. YALAMANCHILI, AND L. NI (1997), *Interconnection Networks: An Engineering Approach*, IEEE CS Press, Los Alamitos, CA.

FLYNN, M. J. (1966), "Very High Speed Computing Systems," *Proc. IEEE*, Vol. 12, pp. 1901–1909.

FLYNN, M. J., AND K. W. RUDD (1996), "Parallel Architectures," *ACM Computing Surveys*, Vol. 28, No. 1, pp. 67–70.

GILL, S. (1958), "Parallel Programming," *Computer Journal*, Vol. 1, April, pp. 2–10.

GILLETT, R., AND R. KAUFMANN (1997), "Using the Memory Channel Network," *IEEE Micro*, Vol. 17, No. 1, pp 19–25.

GUSTAFSON, J. L. (1988), "Reevaluating Amdahl's Law," *Comm. ACM*, Vol. 31, No. 1, pp. 532–533.

HENNESSY, J. L., AND PATTERSON, D. A. (2003), *Computer Architecture: A Quantitative Approach* 3rd edition, Morgan Kaufmann Publishers, San Francisco, CA.

HOGANSON, K., B. WILKINSON, AND W. H. CARLISLE (1997), "Applications of Rhombic Multiprocessors," *Int.Conf. on Parallel and Distributed Processing Techniques and Applications 1997*, Las Vegas, NV, June 30–July 2.

HOLLAND, J. (1959), "A Universal Computer Capable of Executing an Arbitrary Number of Subprograms Simultaneously," *Proc. East Joint Computer Conference*, Vol. 16, pp. 108–113.

KERMANI, P., AND L. KLEINROCK (1979), "Virtual Cut-Through: A New Communication Switching Technique," *Computer Networks*, Vol. 3, pp. 267–286.

KNUCKLES, C. D. (2001), *Introduction to Interactive Programming on the Internet using HTML and JavaScript*, John Wiley, New York.

LEIGHTON, F. T. (1992), *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*, Morgan Kaufmann, San Mateo, CA.

LEISERSON, C. L. (1985), "Fat-Trees: Universal Networks for Hardware-Efficient Supercomputing," *IEEE Trans. Comput.*, Vol. C-34, No. 10, pp. 892–901.

LI, K. (1986), "Shared Virtual Memory on Loosely Coupled Multiprocessor," Ph.D. thesis, Dept. of Computer Science, Yale University.

MARTIN, R. P., A. M. VAHDAT, D. E. CULLER, AND T. E. ANDERSON (1997), "Effects of Communication Latency, Overhead, and Bandwidth in a Cluster Architecture," *Proc 24th Ann. Int. Symp. Comput. Arch.*, ACM, pp. 85–97.

METCALFE, R., AND D. BOGGS (1976), "Ethernet: Distributed Packet Switching for Local Computer Networks," *Comm. ACM*, Vol. 19, No. 7, pp. 395–404.

MINNICH, R., D. BURNS, AND F. HADY (1995), "The Memory-Integrated Network Interface," *IEEE Micro*, Vol. 15, No. 1, pp. 11–20.

NI, L. M., AND P. K. MCKINLEY (1993), "A Survey of Wormhole Routing Techniques in Direct Networks," *Computer*, Vol. 26, No. 2, pp. 62–76.

PACHECO, P. (1997), *Parallel Programming with MPI*, Morgan Kaufmann, San Francisco, CA.

SEITZ, C. L. (1985), "The Cosmic Cube," *Comm. ACM*, Vol. 28, No. 1, pp. 22–33.

SINGH, J. P., J. L. HENNESSY, AND A. GUPTA (1993), "Scaling Parallel Programs for Multiprocessors: Methodology and Examples," *Computer*, Vol. 26, No. 7, pp. 43–50.

STERLING, T., editor (2002a), *Beowulf Cluster Computing with Windows*, MIT Press, Cambridge, MA.

STERLING, T., editor (2002b), *Beowulf Cluster Computing with Linux*, MIT Press, Cambridge, MA.

WILLIAMS, R. (2001), *Computer Systems Architecture A Networking Approach*, Addison-Wesley, Harlow, England.

WILKINSON, B. (1990), "Cascaded Rhombic Crossbar Interconnection Networks," *Journal of Parallel and Distributed Computing*, Vol. 10, No. 1, pp. 96–101.

WILKINSON, B. (1991), "Comparative Performance of Overlapping Connectivity Multiprocessor Interconnection Networks," *Computer Journal*, Vol. 34, No. 3, pp. 207–214.

WILKINSON, B. (1991), "Multiple Bus Network with Overlapping Connectivity," *IEE Proceedings Pt. E: Computers and Digital Techniques*, Vol. 138, No. 4, pp. 281–284.

WILKINSON, B. (1992a), "Overlapping Connectivity Interconnection Networks for Shared Memory Multiprocessor Systems," *Journal of Parallel and Distributed Computing*, Vol. 15, No. 1, pp. 49–61.

WILKINSON, B. (1992b), "On Crossbar Switch and Multiple Bus Interconnection Networks with Overlapping Connectivity," *IEEE Transactions on Computers*, Vol. 41, No. 6, pp. 738–746.

WILKINSON, B. (1996), *Computer Architecture Design and Performance*, 2nd ed., Prentice Hall, London.

WILKINSON, B., AND H. R. ABACHI (1983), "Cross-bar Switch Multiple Microprocessor System," *Microprocessors and Microsystems*, Vol. 7, No. 2, pp. 75–79.

WILKINSON, B., AND J. M. FARMER (1994), "Reflective Interconnection Networks," *Computers and Elect. Eng.*, Vol. 20, No. 4, pp. 289–308.

PROBLEMS

1-1. A multiprocessor consists of 100 processors, each capable of a peak execution rate of 2 Gflops. What is the performance of the system as measured in Gflops when 10% of the code is sequential and 90% is parallelizable?

1-2. Is it possible to have a system efficiency (E) of greater than 100%? Discuss.

1-3. Combine the equation for Amdahl's law with the superlinear speedup analysis in Section 1.2.1 to obtain an equation for speedup given that some of a search has to be done sequentially.

1-4. Identify the host names, IP addresses, and MAC addresses on your system. Determine the IPv4 or IPv6 format used for the network.

1-5. Identify the class of each of the following IPv4 addresses:

(i) 152.66.2.3

(ii) 1.2.3.4

(iii) 192.192.192.192

(iv) 247.250.0.255

given only that class A starts with a 0, class B starts with the pattern 10, class C starts with the pattern 110, and class D starts with the pattern 1110 (i.e., without reference to Figure 1.22).

1-6. Suppose the assigned (IPv4) network address is 153.78.0.0 and it is required to have 6 subnetworks each having 250 hosts. Identify the class of the network address and division of the addresses for the sub-network and hosts. Two addresses must be set aside for the server node.

1-7. A cluster of 32 computers is being set up. The server node has two Ethernet connections, one to the Internet and one to the cluster. The Internet IP address is 216.123.0.0. Devise an IP address assignment for the cluster using C class format.

1-8. A company is proposing an IPv8 format using 512 bits. Do you think this is justified? Explain.

1-9. It is possible to construct a system physically that is a hybrid of a message-passing multicomputer and a shared memory multiprocessor. Write a report on how this might be achieved and its relative advantages over a pure message-passing system and a pure shared memory system.

1-10. (Research project) Write a report on the prospects for a truly incrementally scalable cluster computer system which can accept faster and faster processors without discarding older ones. The concept is to start with a system with a few state-of-the art processors and add a few newer processors each year. Each subsequent year, the processors available naturally get better. At some point the oldest are discarded, but one keeps adding processors. Hence, the system never gets obsolete, and the older processors left still provide useful service. The key issue is how to design the system architecture to accept faster processors and faster interconnects. Another issue is when to discard older processors. Perform an analysis on the best time to discard processors and interconnects.

Chapter 2

Message-Passing Computing

In this chapter, we outline the basic concepts of message-passing computing. The structure of message-passing programs is introduced and how to specify message-passing between processes. We discuss these first in general, and then outline one specific system, MPI (message-passing interface).[1] Finally, we discuss how to evaluate message-passing parallel programs, both theoretically and in practice.

2.1 BASICS OF MESSAGE-PASSING PROGRAMMING

2.1.1 Programming Options

Programming a message-passing multicomputer can be achieved by

1. Designing a special parallel programming language
2. Extending the syntax/reserved words of an existing sequential high-level language to handle message-passing
3. Using an existing sequential high-level language and providing a library of external procedures for message-passing

There are examples of all three approaches. Perhaps the only common example of a special message-passing parallel programming language is the language called *occam*, which was designed to be used with the unique message-passing processor called the *transputer*

[1] Web-based materials for this book include support for two systems, MPI and PVM.

(Inmos, 1984). There are several examples of language extensions for parallel programming, although most, such as High Performance Fortran (HPF), are more geared toward shared memory systems (see Chapter 8). One example of a language extension with explicit message-passing facilities is Fortran M (Foster, 1995).

It is also possible to use a special parallelizing compiler to convert a program written in a sequential programming language, such as Fortran, into executable parallel code. This option was proposed many years ago but is not usually practical for message-passing because traditional sequential programming languages alone do not have the concept of message-passing. Parallelizing compilers are considered briefly in Chapter 8 in the context of shared memory programming.

Here we will concentrate upon the option of programming by using a normal high-level language such as C, augmented with message-passing library calls that perform direct process-to-process message-passing. In this method, it is necessary to say explicitly what processes are to be executed, when to pass messages between concurrent processes, and what to pass in the messages. In this form of programming a message-passing system, we need:

1. A method of creating separate processes for execution on different computers
2. A method of sending and receiving messages

2.1.2 Process Creation

Before continuing, let us reiterate the concept of a *process*. In Chapter 1, the term *process* was introduced for constructing parallel programs. In some instances, especially when testing a program, more than one process may be mapped onto a single processor. Usually, this will not produce the fastest execution speed, as the processor must then time-share between the processes given to it, but it allows a program to be verified before executing the program on a multiple-processor system. There is one situation in which it may be desirable to construct a program to have more than one process on one processor: in order to hide network latencies (this will be discussed in Section 2.3.1). Nevertheless, we will assume that one process is mapped onto each processor and use the term *process* rather than *processor* unless it is necessary to highlight the operation of the processor. First, it is necessary to create processes and begin their execution.

Two methods of creating processes are:

- Static process creation
- Dynamic process creation

In *static process creation*, all the processes are specified before execution and the system will execute a fixed number of processes. The programmer usually explicitly identifies the processes or programs prior to execution by command-line actions. In *dynamic process creation*, processes can be created and their execution initiated during the execution of other processes. Process creation constructs or library/system calls are used to create processes. Processes can also be destroyed. Process creation and destruction may be done conditionally, and the number of processes may vary during execution. Clearly, dynamic process creation is a more powerful technique than static process creation, but it does incur

very significant overhead when the processes are created. The term *process creation* is somewhat misleading because in all cases the code for the processes has to be written and compiled prior to the execution of any process.

In most applications, the processes are neither all the same nor all different; usually there is one controlling process, a "master process," and the remainder are "slaves," or "workers," which are identical in form, only differentiated by their process identification (ID). The process ID can be used to modify the actions of the process or compute different destinations for messages. The processes are defined by programs written by the programmer.

The most general programming model is the *multiple-program, multiple-data* (MPMD) model, in which a completely separate and different program is written for each processor, as shown in Figure 2.1. However, as we have mentioned, normally it is sufficient to have just two different programs, a master program and a slave program. One processor executes the master program, and multiple processors execute identical slave programs. Usually, even though the slave programs are identical, process IDs may be used to customize the execution—for example, to specify the destination of generated messages.

For static process creation especially, the so-called *single-program, multiple-data* (SPMD) model is convenient. In the SPMD model, the different programs are merged into one program. Within the program are control statements that will select different parts for each process. After the source program is constructed with the required control statements to separate the actions of each processor, the program is compiled into executable code for each processor, as illustrated in Figure 2.2. Each processor will load a copy of this code into its local memory for execution, and all processors can start executing their code together. If the processors are of different types, the source code has to be compiled into executable code for each processor type, and the correct type must be loaded for execution by each processor. We will describe the SPMD programming in more detail later (Section 2.2.2), as it is the main approach for one of the most common message-passing systems, MPI.

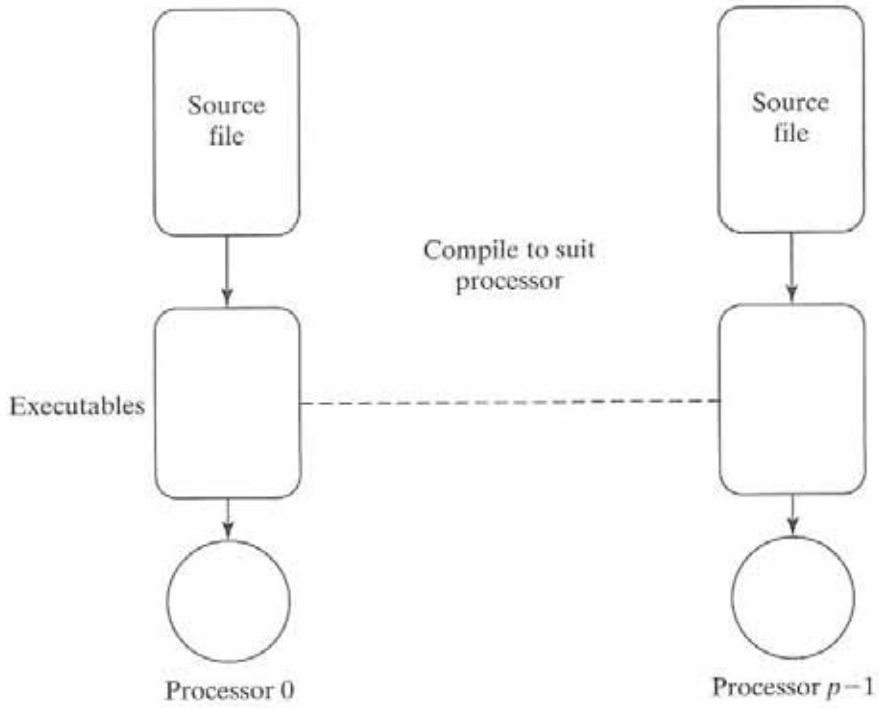


Figure 2.1 Multiple-program, multiple-data (MPMD) model.

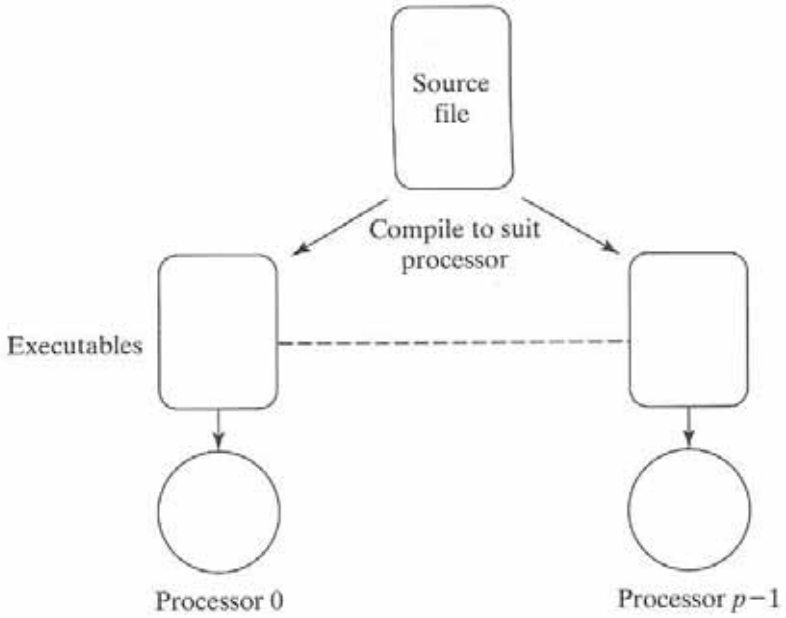


Figure 2.2 Single-program, multiple-data (SPMD) model.

For dynamic process creation, two distinct programs may be written, a master program and a slave program separately compiled and ready for execution. An example of a library call for dynamic process creation might be of the form

```
spawn(name_of_process);
```

which immediately starts another process[2], and both the calling process and the called process proceed together, as shown in Figure 2.3. The process being "spawned" is simply a previously compiled and executable program.

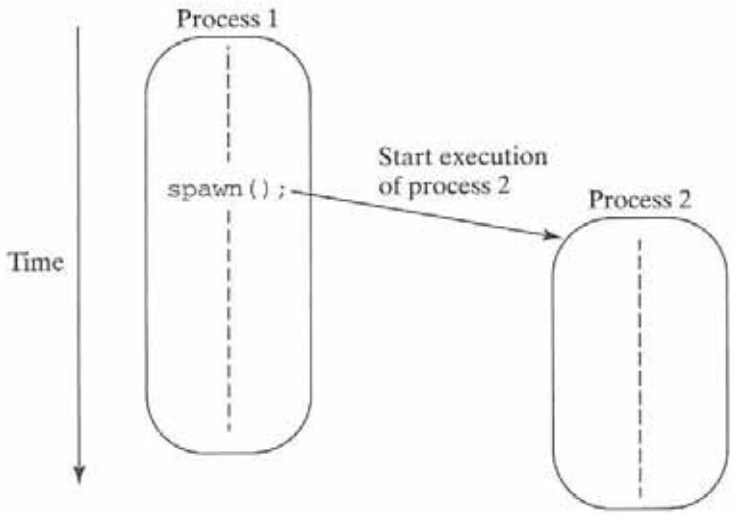


Figure 2.3 Spawning a process.

[2] Courier typeface is used to highlight code, either pseudocode or using a specific language or system.

2.1.3 Message-Passing Routines

Basic Send and Receive Routines. Send and receive message-passing library calls often have the form

```
send(parameter_list)
recv(parameter_list)
```

where `send()` is placed in the source process originating the message, and `recv()` is placed in the destination process to collect the messages being sent. The actual parameters will depend upon the software and in some cases can be complex. The simplest set of parameters would be the destination ID and message in `send()` and the source ID and the name of the location for the receiving message in `recv()`. For the C language, we might have the call

```
send(&x, destination_id);
```

in the source process, and the call

```
recv(&y, source_id);
```

in the destination process, to send the data `x` in the source process to `y` in the destination process, as shown in Figure 2.4. The order of parameters depends upon the system. We will show the process identification after the data and use an `&` with a single data element, as the specification usually calls for a pointer here. In this example, `x` must have been preloaded with the data to be sent, and `x` and `y` must be of the same type and size. Often, we want to send more complex messages than simply one data element, and then a more powerful message formation is needed. The precise details and variations of the parameters of real message-passing calls will be described in Section 2.2, but first we will develop the basic mechanisms. Various mechanisms are provided for send/receive routines for efficient code and flexibility.

Synchronous Message-Passing. The term *synchronous* is used for routines that return when the message transfer has been completed. A synchronous send routine will wait until the complete message that it has sent has been accepted by the receiving process before returning. A synchronous receive routine will wait until the message it is expecting arrives and the message is stored before returning. A pair of processes, one with a synchronous send operation and one with a matching synchronous receive operation, will be synchronized, with neither the source process nor the destination process able to

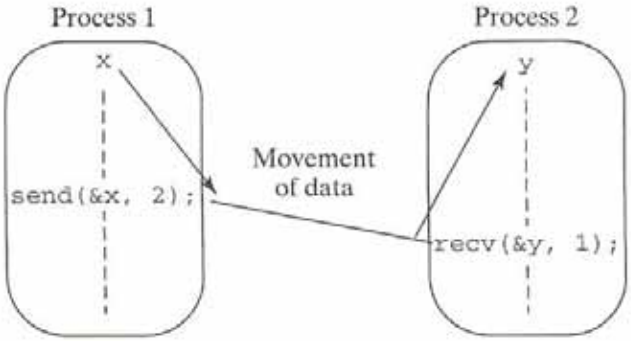


Figure 2.4 Passing a message between processes using `send()` and `recv()` library calls.

proceed until the message has been passed from the source process to the destination process. Hence, synchronous routines intrinsically perform two actions: They transfer data, and they synchronize processes. The term *rendezvous* is used to describe the meeting and synchronization of two processes through synchronous send/receive operations.

Synchronous send and receive operations do not need message buffer storage. They suggest some form of signaling, such as a three-way protocol in which the source first sends a "request to send" message to the destination. When the destination is ready to accept the message, it returns an acknowledgment. Upon receiving this acknowledgment, the source sends the actual message. Synchronous message-passing is shown in Figure 2.5 using the three-way protocol. In Figure 2.5(a), process 1 reaches its `send()` before process 2 has reached the corresponding `recv()`. Process 1 must be suspended in some manner until process 2 reaches its `recv()`. At that time, process 2 must awaken process 1 with some form of "signal," and then both can participate in the message transfer. Note that in Figure 2.5(a), the message is kept in the source process until it can be sent. In Figure 2.5(b), process 2 reaches its `recv()` before process 1 has reached its `send()`. Now, process 2 must be suspended until both can participate in the message transfer. The exact mechanism for suspending and awakening processes is system dependent.

Blocking and Nonblocking Message-Passing. The term *blocking* was formerly also used to describe routines that do not allow the process to continue until the transfer is completed. The routines are "blocked" from continuing. In that sense, the terms *synchronous* and *blocking* were synonymous. The term *nonblocking* was used to describe routines

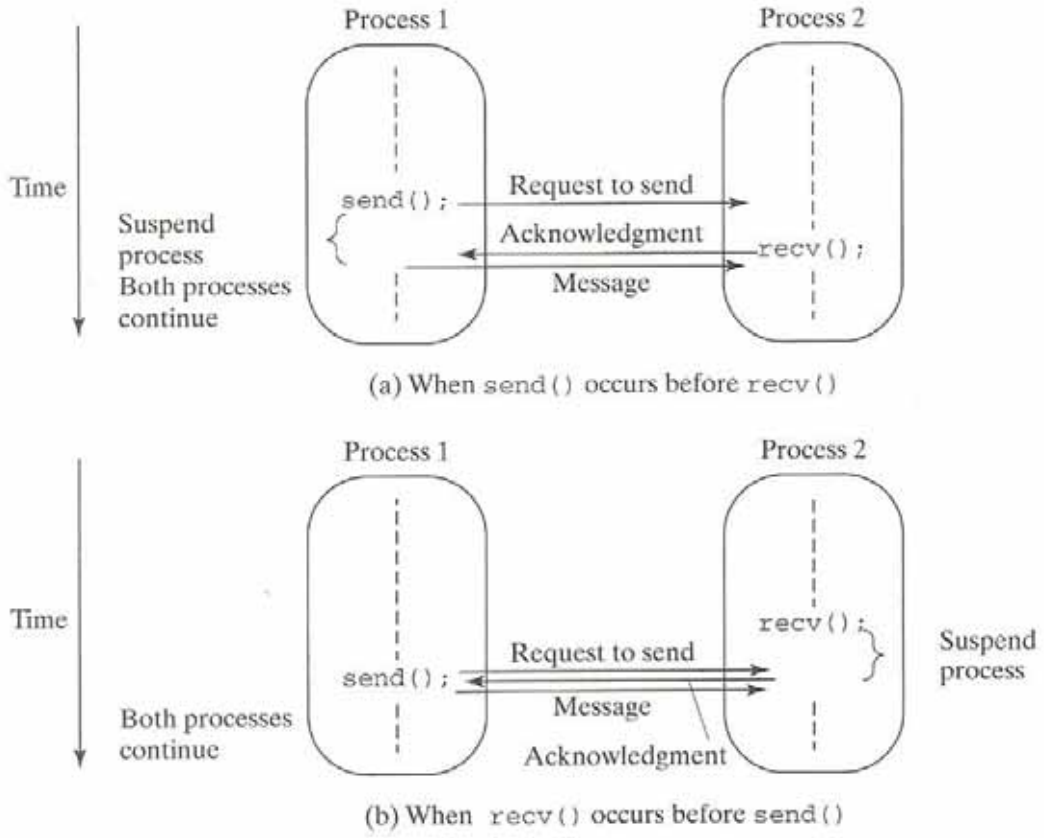


Figure 2.5 Synchronous `send()` and `recv()` library calls using a three-way protocol.

that return whether or not the message had been received. However, the terms *blocking* and *nonblocking* have been redefined in systems such as MPI. We will look into the precise MPI specification later, but for now, let us mention how a send message-passing routine can return before the message transfer has been completed. Generally, a *message buffer* is needed between the source and destination to hold messages, as shown in Figure 2.6. Here, the message buffer is used to hold messages being sent prior to being accepted by `recv()`. For a receive routine, the message has to have been received if we want the message. If `recv()` is reached before `send()`, the message buffer will be empty and `recv()` waits for the message. But for a send routine, once the local actions have been completed and the message is safely on its way, the process can continue with subsequent work. In this way, using such send routines can decrease the overall execution time. In practice, buffers can only be of finite length, and a point could be reached when a send routine is held up because all the available buffer space has been exhausted. It may be necessary to know at some point if the message has actually been received, which will require additional message-passing.

We shall conform to MPI's definitions of terms: Routines that return after their local actions complete, even though the message transfer may not have been completed, are *blocking* or, more accurately, *locally blocking*. Those that return immediately are *nonblocking*. In MPI, nonblocking routines assume that the data storage used for the transfer is not modified by the subsequent statements prior to the data storage being used for the transfer, and it is left to the programmer to ensure this. The term *synchronous* will be used to describe the situation in which the send and receive routines do not return until both occur and the message has been transmitted from the source to the destination. For the most part, *(locally) blocking* and *synchronous* are sufficient for the code in this text.

Message Selection. So far, we have described messages being sent to a specified destination process from a specified source process, where the destination ID is given as a parameter in the send routine and the source ID is given as a parameter in the receive routine. The `recv()` in the destination process will only accept messages from a source process specified as a parameter in `recv()` and will ignore other messages. A special symbol or number may be provided as a *wild card* in place of the source ID to allow the destination to accept messages from any source. For example, the number −1 might be used as a source ID wild card.

To provide greater flexibility, messages can be selected by a *message tag* attached to the message. The *message tag* `msgtag` is typically a user-chosen positive integer (including

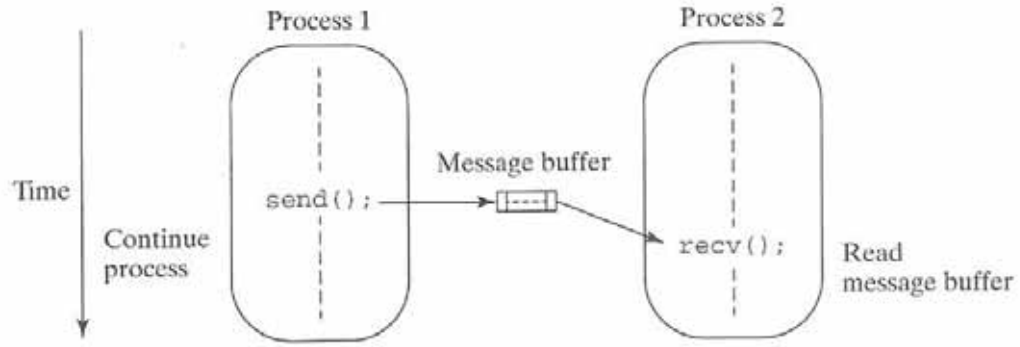


Figure 2.6 Using a message buffer.

zero) that can be used to differentiate between different types of messages being sent. Then specific receive routines can be made to accept only messages with a specific message tag and ignore other messages. A message tag will be an additional parameter in `send()` and `recv()`, usually immediately following the source/destination identification. For example, to send a message, `x`, with message tag 5 from a source process, 1, to a destination process, 2, and assign to `y`, we might have

```
send(&x, 2, 5);
```

in the source process and

```
recv(&y, 1, 5);
```

in the destination process. The message tag is carried within the message. If special type matching is not required, a *wild card* can be used in place of a message tag, so that the `recv()` will match with any `send()`.

The use of message tags is very common. However, it require the programmer to keep track of the message tag numbers used in the program and in any included programs written by others or in library routines that are called. A more powerful message-selection mechanism is needed to differentiate between messages sent within included programs or library routines and those in the user processes. This mechanism will be described later.

Broadcast, Gather, and Scatter. There are usually many other message-passing and related routines that provide desirable features. A process is frequently required to send the same message to more than one destination process. The term *broadcast* is used to describe sending the same message to all the processes concerned with the problem. The term *multicast* is used to describe sending the same message to a defined group of processes. However, this differentiation will not be used here, so it will simply be called broadcast in either case.

Broadcast is illustrated in Figure 2.7. The processes that are to participate in the broadcast must be identified, typically by first forming a named group of processes to be used as a parameter in the broadcast routines. In Figure 2.7, process 0 is identified as the

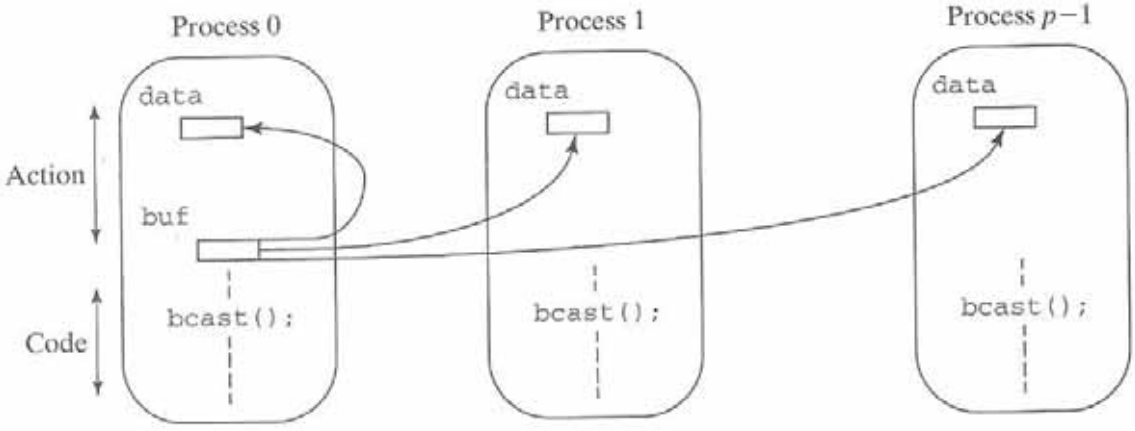


Figure 2.7 Broadcast operation.

root process within the broadcast parameters. The root process could be any process in the group. In this example, the root process holds the data to be broadcast in `buf`. Figure 2.7 shows each process executing the same `bcast()` routine, which is very convenient for the SPMD model, in which all the processes have the same program. Figure 2.7 also shows the root receiving the data, which is the arrangement used in MPI but it depends upon the message-passing system. For the MPMD model, an alternative arrangement is for the source to execute a broadcast routine and destination processes to execute regular message-passing receive routines. In this event, the root process would not receive the data, which is not necessary anyway since the root process already has the data.

As described, the broadcast action does not occur until all the processes have executed their broadcast routine. The broadcast operation will have the effect of synchronizing the processes. The actual implementation of the broadcast will depend upon the software and the underlying architecture. We will look at the implementation later in this chapter and in subsequent chapters. It is important to have an efficient implementation, given the widespread use of broadcast in programs.

The term *scatter* is used to describe sending each element of an array of data in the root to a separate process. The contents of the ith location of the array are sent to the ith process. Whereas broadcast sends the same data to a group of processes, scatter distributes different data elements to processes. Both are common requirements at the start of a program to send data to slave processes. Scatter is illustrated in Figure 2.8. As with broadcast, a group of processes needs to be identified as well as the root process. In this example, the root process also receives a data element. Figure 2.8 shows each process executing the same `scatter()` routine, which again is convenient for the SPMD model.

The term *gather* is used to describe having one process collect individual values from a set of processes. Gather is normally used after some computation has been done by these processes. Gather is essentially the opposite of scatter. The data from the ith process is received by the root process and placed in the ith location of the array set aside to receive the data. Gather is illustrated in Figure 2.9. Process 0 in this example is the root process for the gather. In this example, data in the root is also gathered into the array.

Sometimes the gather operation can be combined with a specified arithmetic or logical operation. For example, the values could be gathered and then added together by the

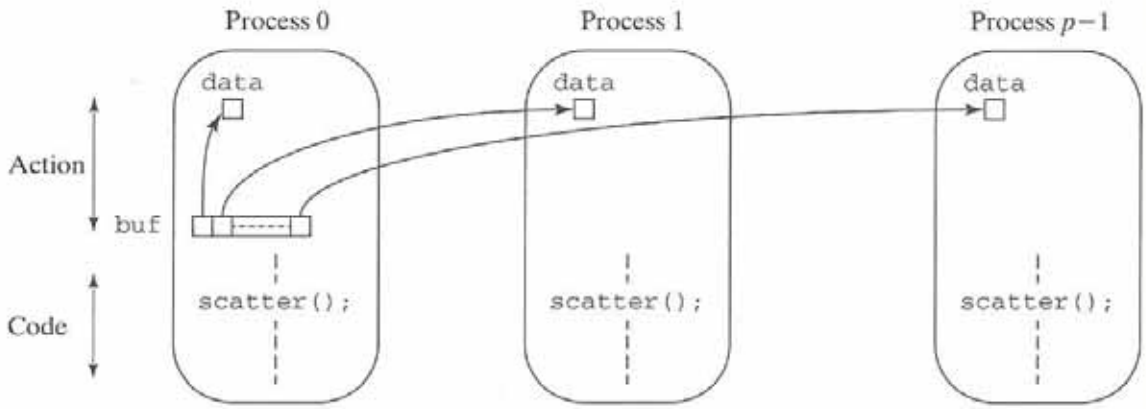


Figure 2.8 Scatter operation.

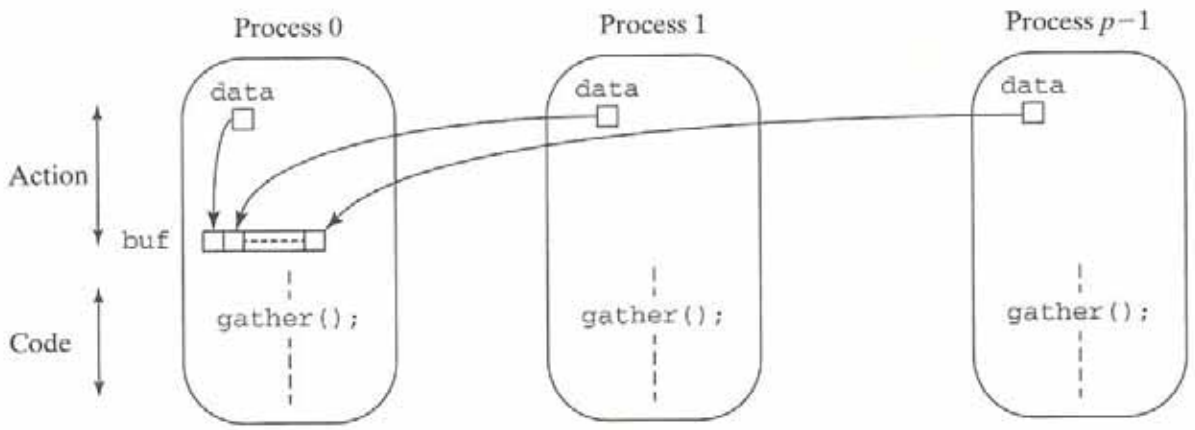


Figure 2.9 Gather operation.

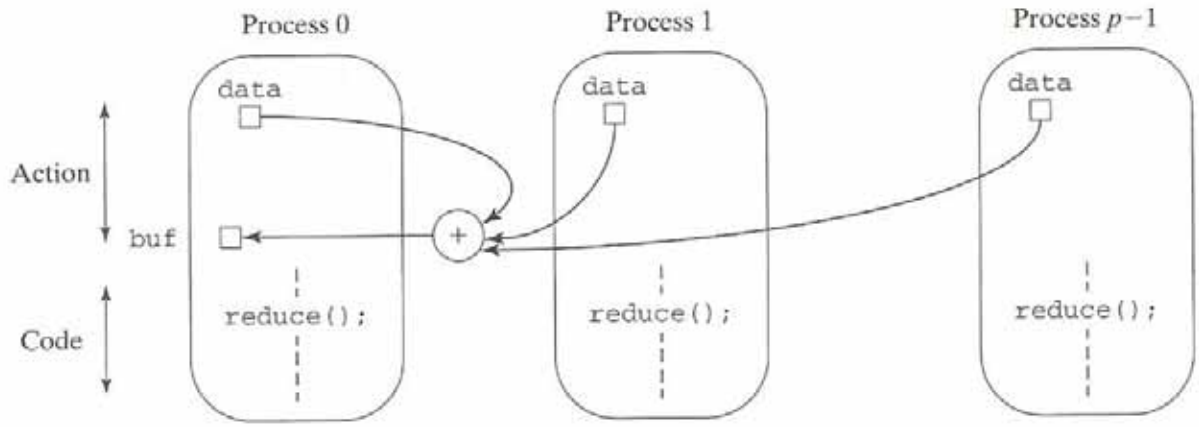


Figure 2.10 Reduce operation (addition).

root, as illustrated in Figure 2.10. Another arithmetic/logical operation could be performed by the root. All such operations are sometimes called *reduce* operations. Most message-passing systems provide for these operations and other related operations. The actual way that the reduce operation is implemented depends upon the implementation. A centralized operation in the root is not the only solution. Partial operations could be distributed among the processes. Whatever the implementation, the idea is to have common collective operations implemented as efficiently as possible.

2.2 USING A CLUSTER OF COMPUTERS

2.2.1 Software Tools

Now let us relate the basic message-passing ideas to a cluster of computers (cluster computing). There have been several software packages for cluster computing, originally described as for networks of workstations. Perhaps the first widely adopted software for using a network of workstations as a multicomputer platform was PVM (parallel virtual machine) developed by Oak Ridge National Laboratories in the late 1980s and used widely in the 1990s. PVM provides a software environment for message-passing between

homogeneous or heterogeneous computers and has a collection of library routines that the user can employ with C or Fortran programs. PVM became widely used, partly because it was made readily available at no charge (from http://www.netlib.org/pvm3). Windows implementations are now available. PVM used dynamic process creation from the start. Apart from PVM, there have also been proprietary message-passing libraries from IBM and others for specific systems. However, it was PVM which made using a network of workstations for parallel programming really practical for most people in the early 1990s.

2.2.2 MPI

To foster more widespread use and portability, a group of academics and industrial partners came together to develop what they hoped would be a "standard" for message-passing systems. They called it MPI (Message-Passing Interface). MPI provides library routines for message-passing and associated operations. A fundamental aspect of MPI is that it defines a standard but not the implementation, just as programming languages are defined but not how the compilers for the languages are implemented. MPI has a large number of routines (over 120 and growing), although we will discuss only a subset of them. An important factor in developing MPI is the desire to make message-passing portable and easy to use. Some changes were also made to correct technical deficiencies in earlier message-passing systems such as PVM. The first version of MPI, version 1.0, was finalized in May 1994 after two years of meetings and discussions. Version 1 purposely omitted some advances that were added to subsequent versions. There have been enhancements to version 1.0 in version1.2. Version 2.0 (MPI-2) was introduced in 1997 and included dynamic process creation, one-sided operations, and parallel I/O.

The large number of functions, even in version 1, is due to the desire to incorporate features that applications programmers can use to write efficient code. However, programs can be written using a very small subset of the available functions. It has been suggested by Gropp, Lusk, and Skjellum (1999a) that successful programs could be written with only six of the 120+ functions. We will mention a few more than just these six "fundamental" functions. Function calls are available for both C and Fortran. We will only consider the C versions. All MPI routines start with the prefix `MPI_` and the next letter is capitalized. Generally, routines return information indicating the success or failure of the call. Such detail is omitted here and can be found in Snir et al. (1998).

Several free implementations of the MPI standard exist, including MPICH from Argonne National Laboratories and Mississippi State University, and LAM from the Ohio Supercomputing Center (now supported by the University of Notre Dame). There are also numerous vender implementations, from Hewlett-Packard, IBM, SGI, SUN, and others. Implementation for Windows clusters exist. A list of MPI implementations and their sources can be found at http://www.osc.edu/mpi/ and http://www.erc.msstate.edu/misc/mpi/implementations.html. A key factor in choosing an implementation is continuing support, because a few early implementations are now not supported at all. A good indicator of support is whether the implementation includes features of the most recent version of the MPI standard (currently MPI-2). The features of available implementations can be found at http://www.erc.msstate.edu/misc/mpi/implementations.html (24 implementations listed here at the time of writing). Most, if not all, implementations do not include every feature of MPI-2. For example, at the time of writing, MPICH did not support MPI one-sided

communication at all. The extended collective operations of MPI-2 are only supported in two implementations (commercial implementations from Hitachi and NEC Corporation). Not having full support can be problematic for writing state-of-the art programs, especially one-sided communication, which is a useful feature.

Process Creation and Execution. As with parallel programming in general, parallel computations are decomposed into concurrent processes. Creating and starting MPI processes is purposely not defined in the MPI standard and will depend upon the implementation. Only static process creation was supported in MPI version 1. This means that all the processes must be defined prior to execution and started together. MPI version 2 introduced dynamic process creation as an advanced feature and has a spawn routine, `MPI_Comm_spawn()`. Even so, one may choose not to use it because of the overhead of dynamic process creation.

Using the SPMD model of computation, one program is written and executed by multiple processors. The way that different programs are started is left to the implementation. Typically, an executable MPI program will be started on the command line. For example, the same executable might be started on four separate processors simultaneously by

```
mpirun prog1 -np 4
```

or

```
prog1 -np 4
```

These commands say nothing about where the copies of `prog1` will be executed. Again, mapping processes onto processors is not defined in the MPI standard. Specific mapping may be available on the command line or by the use of a file holding the names of the executables and the specific processors to run each executable. MPI has support for defining topologies (meshes, etc.), and hence it has the potential for automatic mapping.

Before any MPI function call, the code must be initialized with `MPI_Init()`, and after all MPI function calls, the code must be terminated with `MPI_Finalize()`. Command-line arguments are passed to `MPI_Init()` to allow MPI setup actions to take place. For example,

```
main (int argc, char *argv[])
{
   MPI_Init(&argc, &argv);                    /* initialize MPI */
      .
      .
      .
   MPI_Finalize();                            /* terminate MPI */
}
```

(As in sequential C programs, `&argc`, argument count, provides the number of arguments, and `&argv`, argument vector, is a pointer to an array of character strings.)

Initially, all processes are enrolled in a "universe" called `MPI_COMM_WORLD`, and each process is given a unique rank, a number from 0 to $p-1$, where there are p processes. In MPI terminology, `MPI_COMM_WORLD` is a *communicator* that defines the *scope* of a communication operation, and processes have ranks associated with the communicator. Other

communicators can be established for groups of processes. For a simple program, the default communicator, MPI_COMM_WORLD, is sufficient. However, the concept allows programs, and especially libraries, to be constructed with separate scopes for messages.

Using the SPMD Computational Model. The SPMD model is ideal where each process will actually execute the same code. Normally, though, one or more processors in all applications need to execute different code. To facilitate this within a single program, statements need to be inserted to select which portions of the code will be executed by each processor. Hence, the SPMD model does not preclude a master-slave approach, but both the master code and the slave code must be in the same program. The following MPI code segment illustrates how this could be achieved:

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

   MPI_Comm_rank(MPI_COMM_WORLD, &myrank);        /* find process rank */
   if (myrank == 0)
      master();
   else
      slave();

   MPI_Finalize();
}
```

where `master()` and `slave()` are procedures to be executed by the master process and slave process, respectively. The approach could be used for more than two code sequences. The SPMD model would be inefficient in memory requirements if each processor were to execute completely different code, but fortunately this is unlikely to be required. One advantage of the SPMD model is that command-line arguments can be passed to each process.

Given the SPMD model, any global declarations of variables will be duplicated in each process. Variables that are not to be duplicated could be declared locally; that is, declared within code executed only by that process. For example,

```
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);/* find process rank */
if (myrank == 0) {              /* process 0 actions/local variables */
   int x, y;
      .
      .
      .
} else if (myrank == 1) {       /* process 1 actions/local variables */
   int x, y;
      .
      .
      .
}
```

Here, `x` and `y` in process 0 are different local variables from `x` and `y` in process 1. However such declarations are not favored in C because the scope of a variable in C is from its

declaration to the end of the program or function rather than from the declaration to the end of the current block, which one would want to achieve by declaring the variables within a block. In most instances, one would declare all the variables at the top of the program, and these are then duplicated for each process and essentially are local variables to each process.

Message-Passing Routines. Message-passing communications can be a source of erroneous operation. An intent of MPI is to provide a *safe* communication environment. An example of unsafe communication is shown in Figure 2.11. In this figure, process 0 wishes to send a message to process 1, but there is also message-passing between library routines, as shown. Even though each `send`/`recv` pair has matching source and destination, incorrect message-passing occurs. The use of wild cards makes incorrect operation or deadlock even more likely. Suppose that in one process a nonblocking receive has wild cards in both the tag and source fields. A pair of other processes call library routines that require message-passing. The first send in this library routine may match with the nonblocking receive that is using wild cards, causing erroneous actions.

Communicators are used in MPI for all point-to-point and collective MPI message-passing communications. A communicator is a *communication domain* that defines a set of

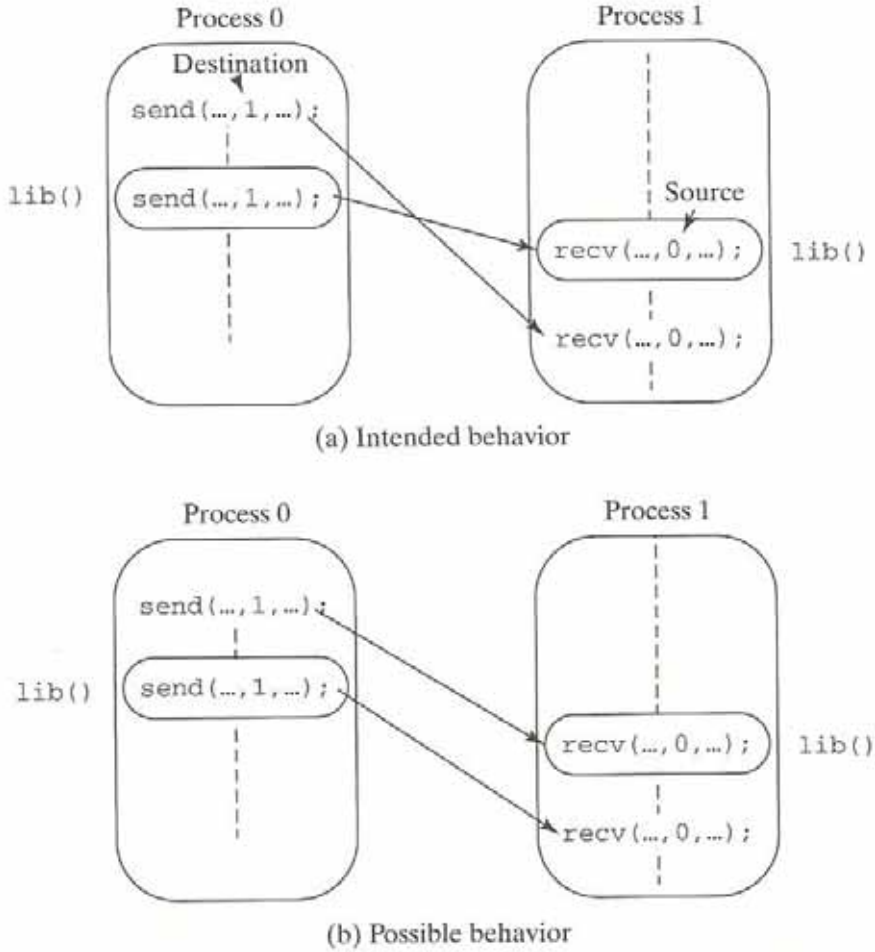


Figure 2.11 Unsafe message-passing with libraries.

processes that are allowed to communicate with one another. In this way, the communication domain of the library can be separated from that of a user program. Each process has a rank within the communicator, an integer from 0 to $p - 1$, where there are p processes. Two types of communicators are available, an *intracommunicator* for communicating within a group, and an *intercommunicator* for communication between groups. A *group* is used to define a collection of processes for these purposes. A process has a unique *rank* in a group (an integer from 0 to $m - 1$, where there are m processes in the group), and a process could be a member of more than one group. For simple programs, only intracommunicators are used, and the additional concept of a group is unnecessary.

A default intracommunicator, `MPI_COMM_WORLD`, exists as the first communicator for all the processes in the application. In simple applications, it is not necessary to introduce new communicators. `MPI_COMM_WORLD` can be used for all point-to-point and collective operations. New communicators are created based upon existing communicators. A set of MPI routines exists for forming communicators from existing communicators (and groups from existing groups); see Appendix A.

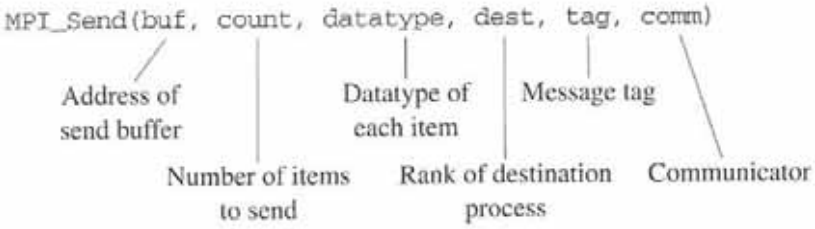
Point-to-Point Communication. Message-passing is done by the familiar send and receive calls. Message tags are present, and wild cards can be used in place of the tag (`MPI_ANY_TAG`) and in place of the source ID in receive routines (`MPI_ANY_SOURCE`).

The datatype of the message is defined in the send/receive parameters. The datatype can be taken from a list of standard MPI datatypes (`MPI_INT`, `MPI_FLOAT`, `MPI_CHAR`, etc.) or can be user created. User-defined datatypes are derived from existing datatypes. In this way, a data structure can be created to form a message of any complexity. For example, a structure could be created consisting of two integers and one float if that is to be sent in one message. Apart from eliminating the need for packing/unpacking routines, as found in the earlier PVM message-passing system, declared datatypes have the advantage that the datatype can be reused. Also, explicit send and receive buffers are not required. This is very useful in reducing the storage requirements of large messages; messages are not copied from the source location to an explicit send buffer. Copying to an explicit send buffer would incur twice the storage space as well as time penalties. (MPI does provide routines for explicit buffers if required.)

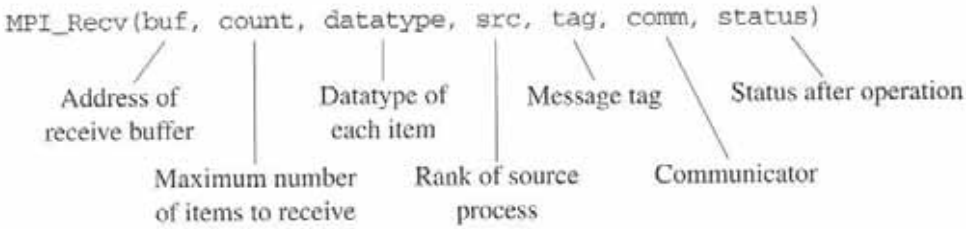
Completion. There are several versions of send and receive. The concepts of being *locally complete* and *globally complete* are used in describing the variations. A routine is locally complete if it has completed all of its part in the operation. A routine is globally complete if all those involved in the operation have completed their parts of the operation and the operation has taken place in its entirety.

Blocking Routines. In MPI, blocking send or receive routines return when they are locally complete. The local completion condition for a blocking send routine is that the location used to hold the message can be used again or altered without affecting the message being sent. A blocking send will send the message and return. This does not mean that the message has been received, just that the process is free to move on without adversely affecting the message. Essentially the source process is blocked for the minimum time that is required to access the data. A blocking receive routine will also return when it is locally complete, which in this case means that the message has been received into the destination location and the destination location can be read.

The general format of parameters of the blocking send is



The general format of parameters of the blocking receive is



Note that a maximum message size is specified in `MPI_Recv()`. If a message is received that is larger than the maximum size, an overflow error occurs. If the message is less than the maximum size, the message is stored at the front of the buffer and the remaining locations are untouched. Usually, though, we would expect to send messages of a known size.

Example

To send an integer x from process 0 to process 1,

```
int x;
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);          /* find process rank */
if (myrank == 0) {
   MPI_Send(&x, 1, MPI_INT, 1, msgtag, MPI_COMM_WORLD);
} else if (myrank == 1) {
   MPI_Recv(&x, 1, MPI_INT, 0, msgtag, MPI_COMM_WORLD, status);
}
```

Nonblocking Routines. A nonblocking routine returns immediately; that is, allows the next statement to execute, whether or not the routine is locally complete. The nonblocking send, `MPI_Isend()`, where `I` refers to the word *immediate*, will return even before the source location is safe to be altered. The nonblocking receive, `MPI_Irecv()`, will return even if there is no message to accept. The formats are

```
MPI_Isend(buf, count, datatype, dest, tag, comm, request)
MPI_Irecv(buf, count, datatype, source, tag, comm, request)
```

Completion can be detected by separate routines, `MPI_Wait()` and `MPI_Test()`. `MPI_Wait()` waits until the operation has actually completed and will return then. `MPI_Test()` returns immediately with a flag set indicating whether the operation has completed at that time.

These routines need to be associated with a particular operation, which is achieved by using the same `request` parameter. The nonblocking receive routine provides the ability for a process to continue with other activities while waiting for the message to arrive.

Example

To send an integer x from process 0 to process 1 and allow process 0 to continue,

```
int x;
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);          /* find process rank */
if (myrank == 0) {
   MPI_Isend(&x, 1, MPI_INT, 1, msgtag, MPI_COMM_WORLD, req1);
   compute();
   MPI_Wait(req1, status);
} else if (myrank == 1) {
   MPI_Recv(&x, 0, MPI_INT, 1, msgtag, MPI_COMM_WORLD, status);
}
```

Send Communication Modes. MPI send routines can have one of four communication modes that define the send/receive protocol. The modes are *standard*, *buffered*, *synchronous*, and *ready*.

In the *standard* mode send, it is not assumed that the corresponding receive routine has started. The amount of buffering, if any, is implementation dependent and not defined by MPI. If buffering is provided, the send could complete before the receive is reached. (If nonblocking, completion occurs when the matching `MPI_Wait()` or `MPI_Test()` returns.)

In the *buffered* mode, send may start and return before a matching receive. It is necessary to provide specific buffer space in the application for this mode. Buffer space is supplied to the system via the MPI routine `MPI_Buffer_attach()` and removed with `MPI_Buffer_detach()`.

In the *synchronous* mode, send and receive can start before each other but can only complete together.

In the *ready* mode, a send can only start if the matching receive has already been reached, otherwise an error will occur. The ready mode must be used with care to avoid erroneous operation.

Each of the four modes can be applied to both blocking and nonblocking send routines. The three nonstandard modes are identified by a letter in the mnemonics (buffered – `b`; synchronous – `s`, and ready – `r`). For example, `MPI_Issend()` is a nonblocking synchronous send routine. This is an unusual combination but has significant uses. The send will return immediately and hence will not directly synchronize the process with the one that has the corresponding receive. The message transfer will presumably complete at some point, which can be determined, as with all immediate mode routines, through the use of `MPI_Wait()` or `MPI_Test()`. This would allow for example to time how long it would take to synchronize processes or to determine whether there is a problem, such as lack of buffer storage. There are some disallowed combinations. Only the standard mode is available for the blocking and nonblocking receive routines, and it is not assumed that the corresponding send has started. Any type of send routine can be used with any type of receive routine.

Collective Communication. Collective communication, such as broadcast, involves a set of processes, as opposed to point-to-point communication involving one source process and one destination process. The processes are those defined by an intracommunicator. Message tags are not present.

Broadcast, Gather, and Scatter Routines. MPI provides a broadcast routine and a range of gather and scatter routines. The communicator defines the collection of processes that will participate in the collection operation. The principal collective operations operating upon data are

```
MPI_Bcast()            - Broadcasts from root to all other processes
MPI_Gather()           - Gathers values for group of processes
MPI_Scatter()          - Scatters buffer in parts to group of processes
MPI_Alltoall()         - Sends data from all processes to all processes
MPI_Reduce()           - Combines values on all processes to single value
MPI_Reduce_scatter()   - Combines values and scatter results
MPI_Scan()             - Computes prefix reductions of data on processes
```

The processes involved are those in the same communicator. There are several variations of the routines. Details of the parameters can be found in Appendix A.

Example

To gather items from the group of processes into process 0, using dynamically allocated memory in the root process, we might use

```
int data[10];                        /*data to be gathered from processes*/
      .
      .
      .

MPI_Comm_rank(MPI_COMM_WORLD, &myrank);              /* find rank */
if (myrank == 0) {
   MPI_Comm_size(MPI_COMM_WORLD, &grp_size);         /*find group size*/
   buf = (int *)malloc(grp_size*10*sizeof(int));     /*allocate memory*/
}
MPI_Gather(data,10,MPI_INT,buf,grp_size*10,MPI_INT,0, MPI_COMM_WORLD);
```

Note that `MPI_Gather()` gathers from all processes, including the root.

Barrier. As in all message-passing systems, MPI provides a means of synchronizing processes by stopping each one until they all have reached a specific "barrier" call. We will look at barriers in detail in Chapter 6 when considering synchronized computations.

Sample MPI Program. Figure 2.12 shows a simple MPI program. The purpose of this program to add a group of numbers together. These numbers are randomly generated and held in a file. The program can be found at http://www.cs.uncc.edu/par_prog and can be used to become familiar with the software environment.

```
#include "mpi.h"
#include <stdio.h>
#include <math.h>
#define MAXSIZE 1000

void main(int argc, char **argv)
{
  int myid, numprocs;
  int data[MAXSIZE], i, x, low, high, myresult, result;
  char fn[255];
  char *fp;

  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD,&myid);

  if (myid == 0) {            /* Open input file and initialize data */
     strcpy(fn,getenv("HOME"));
     strcat(fn,"/MPI/rand_data.txt");
     if ((fp = fopen(fn,"r")) == NULL) {
        printf("Can't open the input file: %s\n\n", fn);
        exit(1);
     }
     for(i = 0; i < MAXSIZE; i++) fscanf(fp,"%d", &data[i]);
  }

/* broadcast data */
   MPI_Bcast(data, MAXSIZE, MPI_INT, 0, MPI_COMM_WORLD);

/* Add my portion Of data */
   x = MAXSIZE/numprocs;           /* must be an integer */
   low = myid * x;
   high = low + x;
   myresult = 0;
   for(i = low; i < high; i++)
      myresult += data[i];
   printf("I got %d from %d\n", myresult, myid);

/* Compute global sum */
   MPI_Reduce(&myresult, &result, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
   if (myid == 0) printf("The sum is %d.\n", result);

   MPI_Finalize();
}
```

Figure 2.12 Sample MPI program.

2.2.3 Pseudocode Constructs

In the preceding sections, we saw specific MPI routines for implementing basic message-passing. Additional code is required for the sometimes numerous parameters, and often many other detailed aspects are involved. For example, error detection code may need to be incorporated. In C, almost all MPI routines return an integer error code in the event of an error;[3] in C++ exception is thrown and error handlers provided. In any

[3] The notable MPI routine that does not have an error code is `MPI_Wtime()` which returns the elapsed time as a double. Presumably this routine cannot cause an error.

event, the code can be identified from a list of error classes and the appropriate action taken. Such additions, although necessary for structurally sound programs, substantially detract from readability. Rather than use real code, we will use a pseudocode for describing algorithms. Our pseudocode will omit the clutter of parameters that are secondary to understanding the code.

The process identification is placed last in the list (as in MPI). To send a message consisting of an integer `x` and a float `y`, from the process called `master` to the process called `slave`, assigning to `a` and `b`, we simply write in the master process

```
send(&x, &y, P_slave);
```

and in the slave process

```
recv(&a, &b, P_master);
```

where `x` and `a` are declared as integers and `y` and `b` are declared as floats. The integer `x` will be copied to `a`, and the float `y` copied to `b`. (Note that we have allowed ourselves the flexibility of specifying more than one data item of different types; in actual code, separate routines may be necessary, or data types created.) We have retained the `&` symbol to indicate that the data parameters are pointers (as they must be for `recv()` at least, and for sending arrays). Where appropriate, the *i*th process will be given the notation P_i, and a tag may be present that would follow the source or destination name. Thus

```
send(&x, P_2, data_tag);
```

sends `x` to process 2, with the message tag `data_tag`. The corresponding receive will have the same tag (or a wide card tag). Sometimes more complex data structures need to be defined, and additional specification is also needed in collective communication routines.

The most common form of basic message-passing routine needed in our pseudocode is the locally blocking `send()` and `recv()`, which will be written as given:

```
send(&data1, P_destination);          /* Locally blocking send */
recv(&data1, P_source);               /* Locally blocking receive */
```

In many instances, the locally blocking versions are sufficient. Other forms will be differentiated with prefixes:

```
ssend(&data1, P_destination);         /* Synchronous send */
```

Virtually all of the code segments given, apart from the message-passing routines, are in the regular C language, although not necessarily in the most optimized or concise manner. For example, for clarity we have refrained from using compressed assignments (e.g., `x += y;`), except for loop counters. Some artistic license has been taken. Exponentiation is written in the normal mathematical way. Generally, initialization of variables is not shown. However, translation of pseudocode to actual message-passing code in MPI or any other message-passing "language" is straightforward.

2.3 EVALUATING PARALLEL PROGRAMS

In subsequent chapters, we will describe various methods of achieving parallelism, and we will need to evaluate these methods. As a prelude to this, let us give a brief overview of the key aspects.

2.3.1 Equations for Parallel Execution Time

Our first concern is how fast the parallel implementation is likely to be. We might begin by estimating the execution time on a single computer, t_s, by counting the computational steps of the best sequential algorithm. For a parallel algorithm, in addition to determining the number of computational steps, we need to estimate the communication overhead. In a message-passing system, the time to send messages must be considered in the overall execution time of a problem. The parallel execution time, t_p, is composed of two parts: a computation part, say t_{comp}, and a communication part, say t_{comm}. Thus we have

$$t_p = t_{comp} + t_{comm}$$

Computational Time. The computation time can be estimated in much the same way as for a sequential algorithm, by counting the number of computational steps. When more than one process is being executed simultaneously, we only need to count the computational steps of the most complex process. Often, all the processes are performing the same operation, so we simply count the number of computation steps of one process. In other situations, we would find the greatest number of computation steps of the concurrent processes. Generally, the number of computational steps will be a function of n and p. Thus

$$t_{comp} = f(n, p)$$

The time units of t_p are those of a computational step. For convenience, we will often break down the computation time into parts separated by message-passing, and then determine the computation time of each part. Then

$$t_{comp} = t_{comp1} + t_{comp2} + t_{comp3} + \cdots$$

where t_{comp1}, t_{comp2}, t_{comp3} ... are the computation times of each part.

Analysis of the computation time usually assumes that all the processors are the same and operating at the same speed. This may be true for a specially designed multicomputer/multiprocessor but may not be true for a cluster. One of the powerful features of clusters is that the computers need not be the same. Taking into account a heterogeneous system would be difficult in a mathematical analysis, so our analysis will assume identical computers. Different types of computers will be taken into account by choosing implementation methods that balance the computational load across the available computers (*load balancing*), as described in Chapter 7.

Communication Time. The communication time will depend upon the number of messages, the size of each message, the underlying interconnection structure, and the mode of transfer. The communication time of each message will depend upon many factors, including network structure and network contention. For a first approximation, we will use

$$t_{comm1} = t_{startup} + wt_{data}$$

for the communication time of a message 1, where $t_{startup}$ is the *startup time*, sometimes called the *message latency*. The startup time is essentially the time needed to send a message with no data. (It could be measured by simply doing that.) It includes the time to pack the message at the source and unpack the message at the destination. The term *latency* is also used to describe a complete communication delay, as in Chapter 1, so we will use the term *startup* time here. The startup time is assumed to be constant. The term t_{data} is the transmission time to send one data word, also assumed to be constant, and there are w data words. The transmission rate is usually measured in bits/second and would be b/t_{data} bits/second when there are b bits in the data word. The equation is illustrated in Figure 2.13. Of course, we do not get such a perfect linear relationship in a real system. Many factors can affect the communication time, including contention on the communication medium. The equation ignores the fact that the source and destination may not be directly linked in a real system so that the message must pass through intermediate nodes. It also assumes that the overhead incurred by including information other than data in the packet is constant and can be part of $t_{startup}$.

The final communication time, t_{comm} will be the summation of the communication times of all the sequential messages from a process. Thus

$$t_{comm} = t_{comm1} + t_{comm2} + t_{comm3} + \cdots$$

where t_{comm1}, t_{comm2}, t_{comm3} ... are the communication times of the messages. (Typically, the communication patterns of all the processes are the same and assumed to take place together, so that only one process need be considered.)

Since the startup and data transmission times, $t_{startup}$ and t_{data}, are both measured in units of one computational step, we can add t_{comp} and t_{comm} together to obtain the parallel execution time, t_p.

Benchmark Factors. Once we have the sequential execution time t_s, the computational time t_{comp}, and the communication time t_{comm}, we can establish the speedup factor and computation/communication ratio described in Chapter 1 for any given algorithm/implementation, namely:

$$\text{Speedup factor} = \frac{t_s}{t_p} = \frac{t_s}{t_{comp} + t_{comm}}$$

$$\text{Computation/communication ratio} = \frac{t_{comp}}{t_{comm}}$$

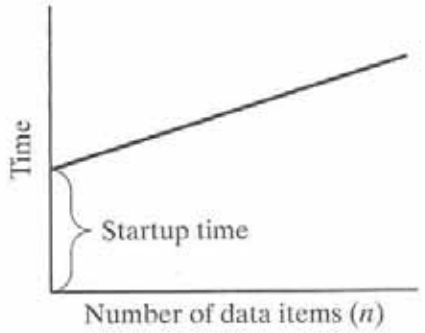


Figure 2.13 Idealized communication time.

Both factors will be functions of the number of processors, p, and the number of data elements, n, and will give an indication of the scalability of the parallel solution with increasing number of processors and increasing problem size. The computation/communication ratio in particular will highlight the effect of communication with increasing problem size and system size.

Important Notes on Interpretation of Equations. There are many assumptions in the analysis given in subsequent chapters, and the analysis is only intended to give a starting point to how an algorithm might perform in practice. The parallel execution time, t_p, will be normalized to be measured in units of an arithmetic operation, which of course will depend upon the computer system. For ease of analysis, it will be assumed that the system is a homogeneous system. Every processor is identical and operating at the same speed. Also, all arithmetic operations are considered to require the same time; for example division requires the same time as addition. Although this is very unlikely in practice, it is a common assumption for analysis. Any additional operations necessary in the formation of the program, such as counting iterations, are not considered.

We will not differentiate between sending an integer and sending a real number, or other formats. All are assumed to require the same time. (This is also not true in practice — in most practical implementations, an 8-bit character will require less time to send than a 64-bit float.) However in many problems, the data type of the data been sent is often the same throughout. The actual startup and transmission times are also dependent upon the computer system and can vary widely between systems. Often the startup time is at least one or two orders of magnitude greater than the transmission time, which is also much greater than the arithmetic operation time. In practice, it is the startup time that will dominate the communication time in many cases. We cannot ignore this term unless n is quite large. (It would, however, be ignored when the equations are converted to order notation; see Section 2.3.2.)

Example

Suppose a computer can operate at a maximum of 1 GFLOPs (10^9 floating point operations per second) and the startup time is 1 μs. The computer could execute 1000 floating point operations in the time taken in the message startup.

Latency Hiding. In the preceding example, one would need 1000 floating point operations between each message just to spend as much time in computing as in message startup. This effect is often cited by shared memory supporters as the Achilles' heel of message-passing multicomputers. One way to ameliorate the situation is to overlap the communication with subsequent computations; that is, by keeping the processor busy with useful work while waiting for the communication to be completed, which is known as *latency hiding*. The nonblocking send routines are provided particularly to enable latency hiding, but even the (locally) blocking send routines allow for subsequent computations to take place while waiting for the destination to receive the message and perhaps return a message. Problem 2-8 explores latency hiding empirically using this approach.

Latency hiding can also be achieved by mapping multiple processes on a processor and using a time-sharing facility that switches from one process to another when the first process is stalled because of incomplete message-passing or for some other reason.

Sometimes the processes are called *virtual processors*. An m-process algorithm implemented on an p-processor machine is said to have a *parallel slackness* of m/p for that machine, where $p < m$. Using parallel slackness to hide latency relies upon an efficient method of switching from one process to another. *Threads* offer an efficient mechanism. See Chapter 8 for further details.

2.3.2 Time Complexity

As with sequential computations, a parallel algorithm can be evaluated through the use of time complexity (notably the O notation — "order of magnitude," "big-oh") (Knuth, 1976). This notation should be familiar from sequential programming and is used to capture characteristics of an algorithm as some variable, usually the data size, tends to infinity. This is especially useful in comparing the execution time of algorithms (*time complexity*) but can also be applied to other computational aspects, such as memory requirements (*space complexity*) as well as speed-up and efficiency in parallel algorithms. Let us first review time complexity as applied to sequential algorithms.

When using the notations for execution time, we start with an estimate of the number of computational steps, considering all the arithmetic and logical operations to be equal and ignoring other aspects of the computation, such as computational tests. An expression of the number of computational steps is derived, often in terms of the number of data items being handled by the algorithm. For example, suppose an algorithm, A1, requires $4x^2 + 2x + 12$ computational steps for x data items. As we increase the number of data items, the total number of operations will depend more and more upon the term $4x^2$. This term will "dominate" the other terms, and eventually the other terms will be insignificant. The growth of the function in this example is *polynomial*. Another algorithm, A2, for the same problem might require $5 \log x + 200$ computational steps.[4] For small x, this has more steps than the first function, A1, but as we increase x, a point will be reached whereby the second function, A2, requires fewer computational steps and will be preferred. In the function $5 \log x + 200$, the first term, $5 \log x$, will eventually dominate the second term, 200, and the second term can be ignored because we only need to compare the dominating terms. The growth of function $\log x$ is *logarithmic*. For a sufficiently large x, logarithmic growth will be less than polynomial growth. We can capture growth patterns in the O notation (big-oh). Algorithm A1 has a big-oh of $O(x^2)$. Algorithm A2 has a big-oh of $O(\log x)$.

Formal Definitions.

O notation. Formally, the O notation can be defined as follows:

$f(x) = O(g(x))$ if and only if there exist positive constants, c and x_0, such that $0 \le f(x) \le cg(x)$ for all $x \ge x_0$

where $f(x)$ and $g(x)$ are functions of x. For example, if $f(x) = 4x^2 + 2x + 12$, the constant $c = 6$ would work with the formal definition to establish that $f(x) = O(x^2)$, since $0 < 4x^2 + 2x + 12 \le 6x^2$ for $x \ge 3$.

[4] Throughout the text, logarithms are assumed to have the base 2 unless otherwise stated, although the base here does not matter.

Unfortunately, the formal definition also leads to alternative functions for $g(x)$ that will also satisfy the definition. For example, $g(x) = x^3$ also satisfies the definition $4x^2 + 2x + 12 \leq 2x^3$ for $x \geq 3$. Normally, we would use the function that grows the least for $g(x)$. In fact, in many cases we have a "tight bound" in that the function $f(x)$ equals $g(x)$ to within a constant factor. This can be captured in the Θ notation.

Θ notation. Formally, the Θ notation can be defined as follows:

> $f(x) = \Theta(g(x))$ if and only if there exist positive constants c_1, c_2, and x_0 such that $0 \leq c_1 g(x) \leq f(x) \leq c_2 g(x)$ for all $x \geq x_0$.

If $f(x) = \Theta(g(x))$, it is clear that $f(x) = O(g(x))$ is also true. One way of satisfying the conditions for the function $f(x) = 4x^2 + 2x + 12$ is illustrated in Figure 2.14. We can actually satisfy the conditions in many ways with $g(x) = x^2$, but can see that $c_1 = 2$, $c_2 = 6$, and $x_0 = 3$ will work; that is, $2x^2 \leq f(x) \leq 6x^2$. Thus, we can say that $f(x) = 4x^2 + 2x + 12 = \Theta(x^2)$, which is more precise than using $O(x^2)$. We should really use the big-oh notation if and only if the upper bound on growth can be satisfied. However, it is common practice to use big-oh in any event.

Ω notation. The lower bound on growth can be described by the Ω notation, which is formally defined as

> $f(x) = \Omega(g(x))$ if and only if there exist positive constants c and x_0 such that $0 \leq cg(x) \leq f(x)$ for all $x \geq x_0$.

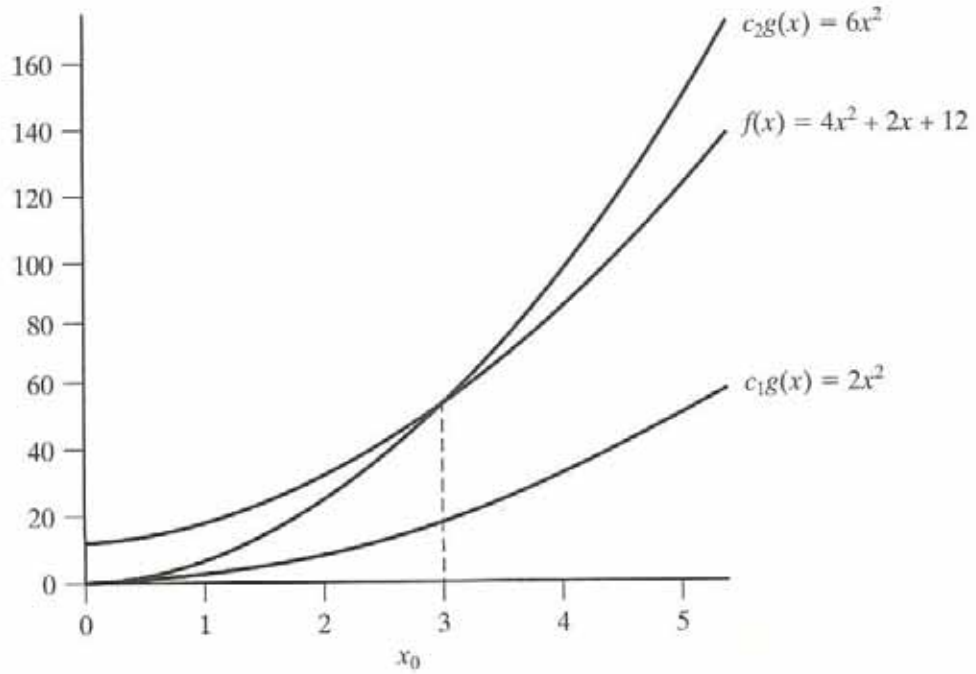


Figure 2.14 Growth of function $f(x) = 4x^2 + 2x + 12$.

It follows from this definition that $f(x) = 4x^2 + 2x + 12 = \Omega(x^2)$ (see Figure 2.14). We can read O() as "grows at most as fast as" and $\Omega()$ as "grows at least as fast as." The function $f(x) = \Theta(g(x))$ is true if and only if $f(x) = \Omega(g(x))$ and $f(x) = O(g(x))$.

The Ω notation can be used to indicate the best-case situation. For example, the execution time of a sorting algorithm often depends upon the original order of the numbers to be sorted. It may be that it requires at least $n \log n$ steps, but could require n^2 steps for n numbers depending upon the order of the numbers. This would be indicated by a time complexity of $\Omega(n \log n)$ and $O(n^2)$.

In this text, we often deal with functions of two variables, p and n. In such cases, the time complexity is also a function of the two variables.

Time Complexity of a Parallel Algorithm. If we use time complexity analysis, which hides lower terms, t_{comm} will have a time complexity of $O(n)$. The time complexity of t_p will be the sum of the complexity of the computation and the communication.

Example

Suppose we were to add n numbers on two computers, where each computer adds $n/2$ numbers together, and the numbers are initially all held by the first computer. The second computer submits its result to the first computer for adding the two partial sums together. This problem has several phases:

1. Computer 1 sends $n/2$ numbers to computer 2.
2. Both computers add $n/2$ numbers simultaneously.
3. Computer 2 sends its partial result back to computer 1.
4. Computer 1 adds the partial sums to produce the final result.

As in most parallel algorithms, there is computation and communication, which we will generally consider separately:

Computation (for steps 2 and 4):

$$t_{comp} = n/2 + 1$$

Communication (for steps 1 and 3):

$$t_{comm} = (t_{startup} + n/2t_{data}) + (t_{startup} + t_{data}) = 2t_{startup} + (n/2 + 1)t_{data}$$

The computational complexity is $O(n)$. The communication complexity is $O(n)$. The overall time complexity is $O(n)$.

Computation/Communication Ratio. Normally, communication is very costly. If both the computation and communication have the same time complexity, increasing n is unlikely to improve the performance. Ideally, the time complexity of the computation should be *greater* than that of the communication, for in that case increasing n will improve the performance. For example, suppose the communication time complexity is $O(n)$ and the computation time complexity is $O(n^2)$. By increasing n, eventually an n can be found that will cause the computation time to dominate the overall execution time. There are notable examples where this can be true. For example, the N-body problem mentioned in Chapter 1, and discussed in Chapter 4, has a communication time complexity of $O(N)$ and a computation time complexity of $O(N^2)$ (using a direct parallel algorithm). This is one of the few problems where the size can be really large.

Cost and Cost-Optimal Algorithms. The *processor-time* product or *cost* (or *work*) of a computation can be defined as

$$\text{Cost} = (\text{execution time}) \times (\text{total number of processors used})$$

The cost of a sequential computation is simply its execution time, t_s. The cost of a parallel computation is $t_p \times p$. A *cost-optimal* parallel algorithm is one in which the cost to solve a problem is proportional to the execution time on a single processor system (using the fastest known sequential algorithm). Thus,

$$\text{Cost} = t_p \times p = k \times t_s$$

where k is a constant. Using time complexity analysis, we can say that a parallel algorithm is cost-optimal algorithm if

Parallel time complexity × number of processors = sequential time complexity

Example

Suppose the best-known sequential algorithm for a problem with n numbers has time complexity of $O(n \log n)$. A parallel algorithm for the same problem that uses p processors and has a time complexity of $O\left(\frac{n}{p}\log n\right)$ is cost-optimal, whereas a parallel algorithm that uses p^2 processors and has time complexity of $O\left(\frac{n^2}{p}\right)$ is not cost-optimal.

2.3.3 Comments on Asymptotic Analysis

Whereas time complexity is widely used for sequential program analysis and for theoretical analysis of parallel programs, the time complexity notation is *much* less useful for evaluating the potential performance of parallel programs. The big-oh and other complexity notations use asymptotic methods (allowing the variable under consideration to tend to infinity), which may not be completely relevant. The conclusions reached from the analyses are based upon the variable under consideration, usually either the data size or the number of processors growing toward infinity. However, often the number of processors is constrained and we are therefore unable to expand the number of processors toward infinity. Similarly, we are interested in finite and manageable data sizes. In addition, the analysis ignores lower terms that could be very significant. For example, the communication time equation

$$t_{\text{comm}} = t_{\text{startup}} + wt_{\text{data}}$$

has a time complexity of $O(w)$, but for reasonable values of w, the startup time would completely dominate the overall communication time. Finally, the analysis also ignores other factors that appear in real computers, such as communication contention.

Shared Memory Programs. Much of our discussion is centered on message-passing programs. For shared memory programs, the communication aspect, of course, does not exist and the time complexity is simply that of the computation, as in a sequential program. In that respect, time complexity might be more relevant. However, an additional aspect of a shared memory program is that the shared data must be accessed in a controlled fashion, causing additional delays. This aspect is considered in Chapter 8.

2.3.4 Communication Time of Broadcast/Gather

Notwithstanding our comments about theoretical analysis, let us look at broadcast/gather operations and their complexity. Almost all problems require data to be broadcast to processes and data to be gathered from processes. Most software environments provide for broadcast and gather. The actual algorithm used will depend upon the underlying architecture of the multicomputer. In the past, the programmer might have been given some knowledge of the interconnection architecture (mesh, etc.) and been able to take advantage of it, but nowadays this is usually hidden from the programmer.

In this text, we concentrate upon using clusters. Again the actual interconnection structure will be hidden from the user, although it is much more likely to provide full simultaneous connections between pairs of computers using switches. In the distant past, Ethernet used a single wire connecting all computers. Broadcast on a single Ethernet connection could be done using a single message that was read by all the destinations on the network simultaneously, (Ethernet protocols provide for this form of communication.) Hence, broadcast could be very efficient and just require a single message:

$$t_{comm} = t_{startup} + wt_{data}$$

with an O(1) time complexity for one data item; for w data items it was $O(w)$.

Of course, most clustered computers will use a variety of network structures, and the convenient broadcast medium available in a single Ethernet will not generally be applicable. Typically messages are sent from the originating computer to multiple destinations, which themselves send the message on to multiple destinations. Once a message arrives at a destination it is converted into a 1-to-N fan-out broadcast call, where the same message is sent to each of the destinations in turn, as shown in Figure 2.15. The same construction will be necessary for gather, except that the messages pass in the opposite direction. In either case, the limiting factor will be whether the messages sent or received are sequential, leading to:

$$t_{comm} = N(t_{startup} + wt_{data})$$

an $O(N)$ communication time complexity for one source connecting to N destinations. We are assuming that the messages at each level occur at the same time (which of course would not happen in practice).

The 1-to-N fan-out broadcast applied to a tree structure is shown in Figure 2.16. The complexity here will depend upon the number of nodes at each level and the number

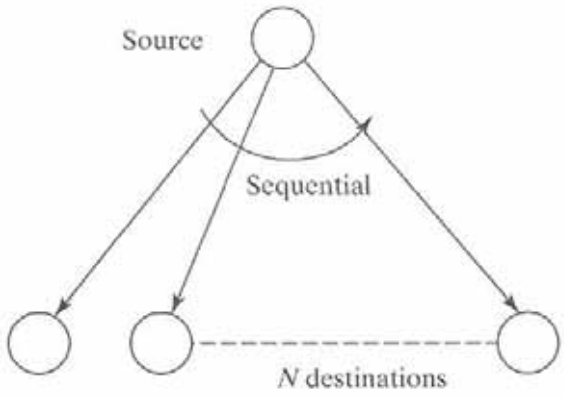


Figure 2.15 1-to-N fan-out broadcast.

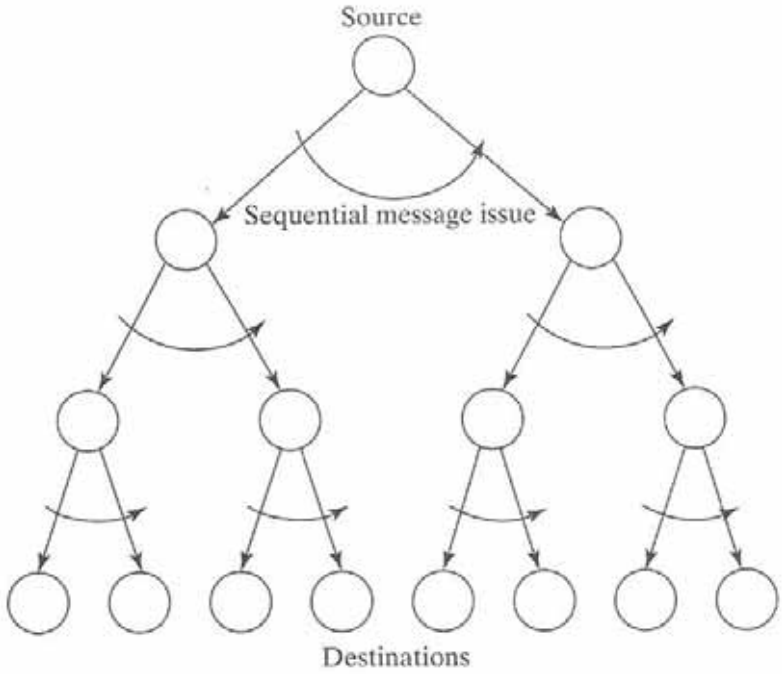


Figure 2.16 1-to-*N* fan-out broadcast on a tree structure.

of levels. For a binary tree N = 2 and log p levels if there are p final destinations. This leads to

$$t_{comm} = 2(\log p)(t_{startup} + wt_{data})$$

assuming again that the messages at each level occur at the same time. (It is left as an exercise to determine the communication time if this assumption is not made.)

One disadvantage of a binary tree implementation of broadcast is that if a node fails in the tree, all the nodes below it will not receive the message. In addition, a binary tree would be difficult to implement in a library call. The programmer can, of course, achieve it with explicit coding in the processes.

2.4 DEBUGGING AND EVALUATING PARALLEL PROGRAMS EMPIRICALLY

In writing a parallel program, we first want to get it to execute correctly. Then we will become interested in how fast the program executes. Finally, we would like to see whether it can be made to execute faster.

2.4.1 Low-Level Debugging

Getting a parallel program to work properly can be a significant intellectual challenge. It is useful to write a sequential version first, based upon the ultimate parallel algorithm. Having said that, the parallel code still has to be made to work. Errors in sequential programs are found by *debugging*. A common way is to instrument the code; that is, to insert code that outputs intermediate calculated values as the program executes. Print statements are used to output the intermediate values. Similar techniques could be used in parallel programs, but this approach has some very significant consequences. First and foremost, instrumenting a sequential program makes it execute slower, but it still functions deterministically and

produces the same answer. Instrumenting a parallel program by inserting code in the different processes will certainly slow down the computations. It also may cause the instructions to be executed in a different interleaved order, as generally each process will be affected differently. It is possible for a nonworking program to start working after the instrumentation code is inserted — which would certainly indicate that the problem lies within interprocess timing.

We should also mention that since processes might be executing on a remote computer, output from print statements may need to be redirected to a file in order to be seen at the local computer. Message-passing software often has facilities to redirect output.

The lowest level of debugging (in desperation) is to use a debugger. Primitive sequential-program debugging tools, such as dbx, exist (but are rarely used) to examine registers and perhaps set "breakpoints" to stop the execution. Applying these techniques to parallel programs would be of little value because of such factors as not knowing the precise interleaved order of events in different processes. One scenario would be to run the debugger on individual processors and watch the output in separate display windows. A parallel computation could have many simultaneous processes, which would make this approach unwieldy. In the case of dynamic process creation, system facilities may be needed to start spawned processes through a debugger.

Parallel computations have characteristics that are not captured by a regular sequential debugger, such as timing of events. Events may be recognized when certain conditions occur. In addition to what might appear in a sequential program, such as access to a memory location, an event in this context may be a message being sent or received. Parallel debuggers are available (McDowell and Helmbold, 1989).

2.4.2 Visualization Tools

Parallel computations lend themselves to visual indication of their actions, and message-passing software often provides visualization tools as part of the overall parallel programming environment. Programs can be watched as they are executed in a *space-time diagram* (or *process-time diagram*). A hypothetical example is shown in Figure 2.17. Each waiting period indicates a process being idle, often waiting for a message to be received. Such visual presentations may help spot erroneous actions. The events that created the space-time diagram can be captured and saved so that the presentation can be replayed without having to reexecute the program. Also of interest is a *utilization-time diagram*, which shows the amount of time spent by each process on communication, waiting, and message-passing library routines. Apart from its help in debugging, the utilization-time diagram also indicates the efficiency of the computation. Finally, animation may be useful where processes are shown in a two-dimensional display and changes of state are shown in a movie form.

Implementations of visualization tools are available for MPI. An example is the Upshot program visualization system (Herrarte and Luske, 1991). All forms of visualization imply software "probes" into the execution, which may alter the characteristics of the computation. It certainly makes the computation proceed much slower. (Hardware performance monitors are possible that do not usually affect the performance. For example, there are some that simply monitor the system bus, but these are not widely deployed.)

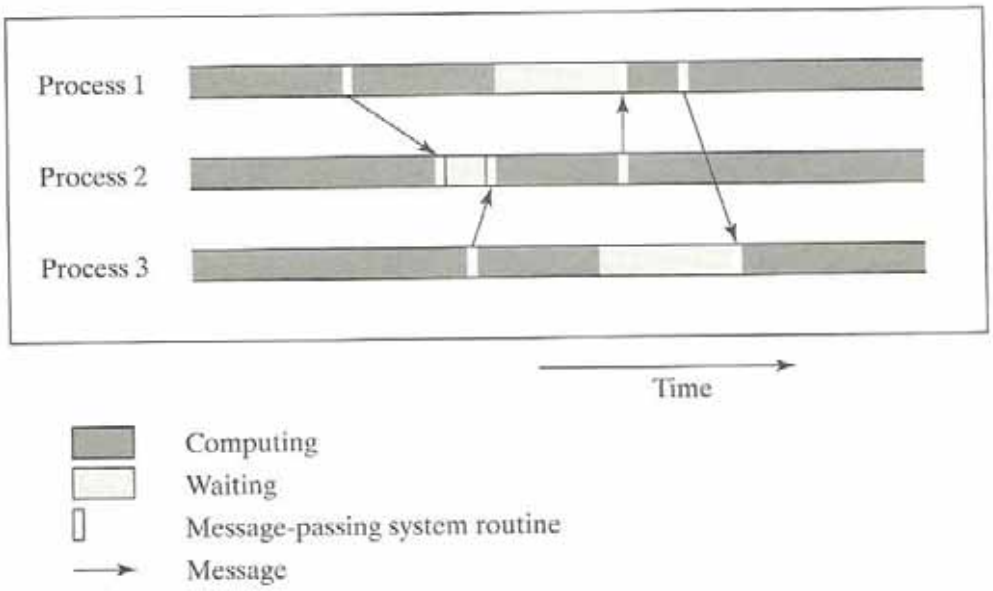


Figure 2.17 Space-time diagram of a parallel program.

2.4.3 Debugging Strategies

Geist et al. (1994a) suggest a three-step approach to debugging message-passing programs:

1. If possible, run the program as a single process and debug as a normal sequential program.
2. Execute the program using two to four multitasked processes on a single computer. Now examine actions, such as checking that messages are indeed being sent to the correct places. Very often mistakes are made with message tags and messages are sent to the wrong places.
3. Execute the program using the same two to four processes but now across several computers. This step helps find problems that are caused by network delays related to synchronization and timing.

Placing error-checking code in the program is always important as good programming practice, but is particularly important in parallel programs to ensure that faulty conditions can be handled and not cause deadlock. Many message-passing routines return an error code if an error is detected. Though not necessary for these routines to execute, error codes should be recognized if they occur. They are also useful in debugging. MPI also can be made to return error codes, but the default situation is for the program to abort when an error is encountered.

2.4.4 Evaluating Programs

Measuring Execution Time. Time-complexity analysis might give an insight into the potential of a parallel algorithm and is useful in comparing different algorithms. However, given our comments in Section 2.3.4 about time-complexity analysis, only when the algorithm is coded and executed on a multiprocessor system will it be truly known how

well the algorithm actually performs. As with low-level debugging (Section 2.4), programs can be instrumented with additional code. To measure the execution time of a program or the *elapsed time* between two points in the code in seconds, we could use regular system calls, such as `clock()`, `time()`, or `gettimeofday()`. Thus, to measure the execution time between point `L1` and point `L2` in the code, we might have a construction such as

```
        .
        .
        .
L1:   time(&t1);                          /* start timer */
        .
        .
        .
L2:   time(&t2);                          /* stop timer */
        .
        .
        .
      elapsed_time = difftime(t2, t1);    /* elapsed_time = t2 - t1 */
      printf("Elapsed time = %5.2f seconds", elapsed_time);
```

Elapsed time will include the time waiting for messages, and it is assumed that the processor is not executing any other program at the same time.

Often the message-passing software itself includes facilities for timing; for example, by providing library calls that return the time or by displaying time on space-time diagrams (as described in Section 2.4). MPI provides the routine `MPI_Wtime()` for returning time in seconds. In general, each processor will be using its own clock, and the time returned will not necessarily be synchronized with the clocks of other processors unless clock synchronization is available. Clock synchronization is defined in MPI as an environment attribute but may not be implemented in a system because it usually incurs a very significant system overhead.

Communication Time by the Ping-Pong Method. Point-to-point communication time of a specific system can be found using the *ping-pong method* as follows. One process, say P_0, is made to send a message to another process, say P_1. Immediately upon receiving the message, P_1 sends the g back to P_0. The time involved in this message communication is recorded at P_0. This time is divided by two to obtain an estimate of the time of one-way communication:

Process P_0

```
        .
        .
        .
L1:   time(&t1);
      send(&x, P1);
      recv(&x, P1);
L2:   time(&t2);
      elapsed_time = 0.5 * difftime(t2, t1);
      printf("Elapsed time = %5.2f seconds", elapsed_time);
        .
        .
        .
```

Process P_1

```
    .
    .
    .
recv(&x, P0);
send(&x, P0);
    .
    .
    .
```

Problem 2-5 explores measuring communication times.

Profiling. A *profile* of a program is a histogram or graph showing the time spent on different parts of the program. A profile can show the number of times certain source statements are executed, as illustrated in Figure 2.18. The *profiler* actually producing the results must capture the information from the executing program and in doing so will affect the execution time. To count the appearance of each instruction would probably be too invasive. Instead, the executing code is usually probed or sampled at intervals to give statistical results. Probing in any form will affect the execution characteristics. This is especially important in parallel programs that have interrelationships between concurrent processes.

Profiling can be used to identify "hot spot" places in a program visited many times during the program execution. These places should be optimized first, a technique that is applicable to both sequential and parallel programs.

2.4.5 Comments on Optimizing Parallel Code

Once the performance has been measured, structural changes may need to be made to the program to improve its performance. Apart from optimizations that apply to regular single-processor programs, such as moving constant calculations to the outside of loops, several parallel optimizations are possible. These usually relate to the architecture of the multiprocessor system. The number of processes can be changed to alter the process granularity. The amount of data in the messages can be increased to lessen the effects of startup times. It may often be better to recompute values locally than to send computed values in additional messages from one process to other processes needing these values. Communication and

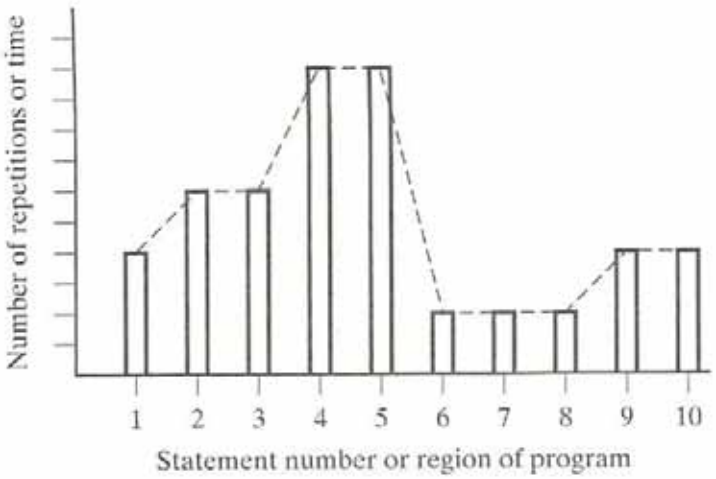


Figure 2.18 Program profile.

computation can be overlapped (latency hiding, Section 2.3.1). One can perform a *critical path analysis* on the program; that is, determine the various concurrent parts of the program and find the longest path that dominates the overall execution time.

A much less obvious factor than those just mentioned is the effect of the memory hierarchy. Processors usually have high-speed cache memory. A processor first accesses its cache memory, and only afterward accesses the main memory if the information is not already in the cache. The information is brought into the cache from the main memory by a previous reference to it, but only a limited amount of data can be held in the cache. The best performance will result if as much of the data as possible resides in the cache when needed. This can sometimes be achieved by a specific strategy for parallelization and sometimes by simply reordering the memory requests in the program. In Chapter 11, some numerical algorithms are presented. Simply performing the sequence of arithmetic operations in a different order can result in more of the data being in the cache when subsequent references to the data occur.

2.5 SUMMARY

This chapter introduced the following concepts:

- Basic message-passing techniques
- Send, receive, and collective operations
- Software tools for harnessing a network of workstations
- Modeling communication
- Communication latency and latency hiding
- Time complexity of parallel algorithms
- Debugging and evaluating parallel programs

FURTHER READING

We have concentrated upon message-passing routines in general and on one system in particular, MPI. Another system that is widely used is PVM (Parallel Virtual Machine). One of the earliest articles on PVM is Sunderam (1990). More details on PVM can be found in Geist et al. (1994a and 1994b). Further details on MPI can be found in Gropp, Lusk, and Skjellum (1999) and Snir et al. (1998). MPI-2 can be found specifically in Gropp et al (1998), and Gropp, Lusk, and Thakur (1999). The home page for the MPI forum, http://www.mpi-forum.org, provides many official documents and information about MPI. Other sources include Dongarra et al. (1996), which gives references to several earlier message-passing systems in addition to PVM. Programming in PVM and MPI is described in several chapters of Sterling (2002a and b). Papers on scattering and gathering messages include Bhatt et al. (1993).

Fundamental design and analysis of sequential algorithms can be found in the classic text by Knuth (1973). Other texts include Aho, Hopcroft, and Ullman (1974). There have

been many others since. A modern, comprehensive, and very well written text on this subject is Cormen, Leiserson, and Rivest (1990), which contains over 1000 pages. Design and analysis of parallel algorithms can be found in several texts, including Akl (1989) and JáJá (1992). A rather unique book that integrates sequential and parallel algorithms is Berman and Paul (1997).

Details of parallel debugging can be found in McDowell and Helmbold (1989) and Simmons et al. (1996). Other papers include Kraemer and Stasko (1993) and Sistare et al. (1994). A special issue of the *IEEE Computer* on parallel and distributed processing tools (November. 1995) contained several useful articles on performance-evaluation tools for parallel systems.

BIBLIOGRAPHY

AHO, A. V., J. E. HOPCROFT, AND J. D. ULLMAN (1974), *The Design and Analysis of Computer Algorithms*, Addison-Wesley Publishing Company, Reading, MA.

AKL, S. G. (1989), *The Design and Analysis of Parallel Algorithms*, Prentice Hall, Englewood Cliffs, NJ.

BERMAN, K. A., AND J. L. PAUL (1997), *Fundamentals of Sequential and Parallel Algorithms*, PWS Publishing Company, Boston, MA.

BHATT, S. N., ET AL. (1993), "Scattering and Gathering Messages in Networks of Processors, *IEEE Trans. Comput.*, Vol. 42, No. 8, pp. 938–949.

CORMEN, T. H., C. E. LEISERSON, AND R. L. RIVEST (1990), *Introduction to Algorithms*, MIT Press, Cambridge, MA.

DONGARRA, J., S. W. OTTO, M. SNIR, AND D. WALKER (1996), "A Message-Passing Standard for MMP and Workstations," *Comm. ACM*, Vol. 39, No. 7, pp. 84–90.

FAHRINGER, T. (1995), "Estimating and Optimizing Performance for Parallel Programs," *Computer*, Vol. 28, No. 11, pp. 47–56.

FOSTER, I. (1995), *Designing and Building Parallel Programs*, Addison-Wesley, Reading, MA.

GEIST, A., A. BEGUELIN, J. DONGARRA, W. JIANG, R. MANCHEK, AND V. SUNDERAM (1994a), *PVM3 User's Guide and Reference Manual*, Oak Ridge National Laboratory, TN.

GEIST, A., A. BEGUELIN, J. DONGARRA, W. JIANG, R. MANCHEK, AND V. SUNDERAM (1994b), *PVM: Parallel Virtual Machine*, MIT Press, Cambridge, MA.

GRAMA, A., A. GUPTA, G. KARYPIS, AND V. KUMAR (2003), *Introduction to Parallel Computing*, 2nd edition, Benjamin/Cummings, Redwood City, CA.

GROPP, W., S. HUSS-LEDERMAN, A. LUMSDAINE, E. LUSK, B. NITZBERG, W. SAPHIR, AND M. SNIR (1998), *MPI, The Complete Reference*, Volume 2, *The MPI-2 Extensions*, MIT Press, Cambridge, MA.

GROPP, W., E. LUSK, AND A. SKJELLUM (1999a), *Using MPI Portable Parallel Programming with the Message-Passing Interface*, 2nd edition, MIT Press, Cambridge, MA.

GROPP, W., E. LUSK, AND RAJEEV THAKUR (1999b), *Using MPI-2 Advanced Features of the Message-Passing Interface*, MIT Press, Cambridge, MA.

HERRARTE, V., AND LUSKE, E. (1991), "Studying Parallel Program Behavior with Upshot," Technical Report ANL-91/15, Mathematics and Computer Science Division, Argonne National Laboratory.

INMOS LTD. (1984), *Occam Programming Manual*, Prentice Hall, Englewood Cliffs, NJ.

JÁJÁ, J. (1992), *An Introduction to Parallel Algorithms*, Addison Wesley, Reading, MA.

KARONIS, N. T. (1992), "Timing Parallel Programs That Use Message-Passing," *J. Parallel & Distributed Computing*, Vol. 14, pp. 29–36.

KNUTH, D. E. (1973), *The Art of Computer Programming*, Addison-Wesley, Reading, MA.

KNUTH, D. E. (1976), "Big Omicron, Big Omega and Big Theta," *SIGACT News* (April–June), pp. 18–24.

KRAEMER, E., AND J. T. STASKO (1993), "The Visualization of Parallel Systems: An Overview," *J. Parallel & Distribut. Computing*, Vol. 18, No. 2 (June), pp. 105–117.

KRONSJO, L. (1985), *Computational Complexity of Sequential and Parallel Algorithms*, Wiley, NY.

MCDOWELL, C. E., AND D. P. HELMBOLD (1989), "Debugging Concurrent Programs," *Computing Surveys*, Vol. 21, No. 4, pp. 593–622.

SIMMONS, M., A. HAYES, J. BROWN, AND D. REED (EDS.) (1996), *Debugging and Performance Tools for Parallel Computing Systems*, IEEE CS Press, Los Alamitos, CA.

SISTARE, S., D. ALLEN, R. BOWKER, K. JOURDENAIS, J. SIMONS, AND R. TITLE (1994), "A Scalable Debugger for Massively Parallel Message-Passing Programs," *IEEE Parallel & Distributed Technology*, (Summer), pp. 50–56.

SNIR, M., S. W. OTTO, S. HUSS-LEDERMAN, D. W. WALKER, AND J. DONGARRA (1998), *MPI - The Complete Reference Volume 1, The MPI Core, 2nd edition*, MIT Press, Cambridge, MA.

STERLING, T., editor (2002a), *Beowulf Cluster Computing with Windows*, MIT Press, Cambridge, MA.

STERLING, T., editor (2002b), *Beowulf Cluster Computing with Linux*, MIT Press, Cambridge, MA.

SUNDERAM, V. (1990), "PVM: A Framework for Parallel Distributed Computing," *Concurrency: Practice & Experience*, Vol. 2, No. 4, pp. 315–339.

PROBLEMS

2-1. Develop an equation for message communication time, t_{comm}, that incorporates a delay through multiple links as would occur in a static interconnection network. Develop the equation for a mesh assuming that all message destinations are randomly chosen.

2-2. Pointers are used in the send and receive used in book and in MPI, but they can be avoided by passing the arguments by value. For example, pointers can be eliminated in the receive routine by having the routing return the message data, which then can be assigned to a variable, i.e., `x = recv(sourceID)`. Write new routines to "wrap" around regular MPI send and receive routines to avoid pointers and demonstrate their use.

2-3. To send a message from a specific source process to a specific destination process, it is necessary for the source process to know the destination TID (task identification) or rank and for the destination to know the source TID or rank. Explain how each process can obtain the TID or rank of the other process in MPI. Give a program example.

2-4. (A suitable first assignment) Compile and run the MPI program to add numbers, as given in Figures 2.14 and 2.16 (or as found in http://www.cs.uncc.edu/par_prog as the "sample program" in the compiling instructions) and execute on your system. Modify the program so that the maximum number is found and output as well as the sum.

2-5. Measure the time to send a message in a parallel programming system by using code segments of the form

Master

```
        .
        .
        .
L1:     time(&t1);
        send(&x, P_slave);
L2:     time(&t2);
        tmaster = difftime(t2, t1);
        recv(&tslave, P_slave);
        printf("Master Time = %d", tmaster);
        printf("Slave Time = %d", tslave);
        .
        .
        .
```

Slave

```
        .
        .
        .
L1:     time(&t1);
        recv(&x, P_master);
L2:     time(&t2);
        tslave = difftime(t2, t1);
        send(&tslave, P_master);
        .
        .
        .
```

Repeat with the ping-pong method described in Section 2.4.4. Experiment with sending groups of multiple messages and messages of different sizes to obtain a good estimate for the time of message transfers. Plot the time for sending a message against the size of the message, and fit a line to the results. Estimate the startup time, $t_{startup}$ (latency), and the time to send one data item, t_{data}.

2-6. Repeat Problem 2-5 for broadcast and other collective routines as available on your system.

2-7. Compare the use of broadcast and gather routines using individual send and receive routines empirically.

2-8. Experiment with latency hiding on your system to determine how much computation is possible between sending messages. Investigate using both nonblocking and locally blocking send routines.

2-9. Develop an equation for communication time and time complexity for the binary tree broadcast described in Section 2.3.4 assuming the messages at each level do not occur at the same time (as would happen in practice). Extend for an m-ary tree broadcast (each node having m destinations).

2-10. If you have both PVM and MPI available (or any two systems), make a comparative study of the communication times on the systems by passing messages between processes that have been instrumented to measure the communication times.

Chapter 3

Embarrassingly Parallel Computations

In this chapter, we will consider the "ideal" computation from a parallel computing standpoint — a computation that can be divided into a number of completely independent parts, each of which can be executed by a separate processor. This is known as an *embarrassingly parallel* computation. We will look at sample embarrassingly parallel computations before moving on in other chapters to computations that do not decompose as well. The material in this chapter can form the basis of one's first parallel program.

3.1 IDEAL PARALLEL COMPUTATION

Parallel programming involves dividing a problem into parts in which separate processors perform the computation of the parts. An ideal parallel computation is one that can be immediately divided into completely independent parts that can be executed simultaneously. This is picturesquely called *embarrassingly parallel* (a term coined by Geoffrey Fox; Wilson, 1995) or perhaps more aptly called *naturally parallel*. Parallelizing these problems should be obvious and requires no special techniques or algorithms to obtain a working solution. Ideally, there would be no communication between the separate processes; that is, a completely disconnected computational graph, as shown in Figure 3.1. Each process requires different (or the same) data and produces results from its input data without any need for results from other processes. This situation will give the maximum possible speedup if all the available processors can be assigned processes for the total duration of the computation. The only constructs required here are simply to distribute the data and to start the processes. Interesting, there are many significant real applications that

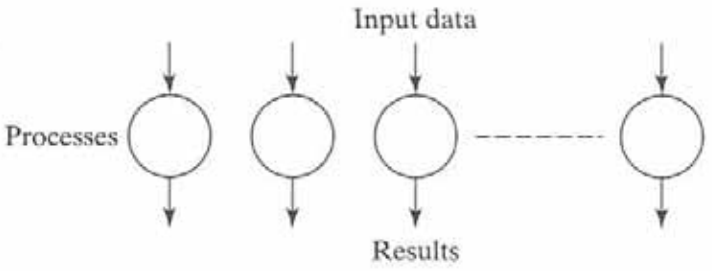


Figure 3.1 Disconnected computational graph (embarrassingly parallel problem).

are embarrassingly parallel, or at least nearly so. Often the independent parts are identical computations and the SPMD (single-program multiple-data) model is appropriate, as suggested in Figure 3.1. The data is not shared, and hence distributed memory multiprocessors or message-passing multicomputers are appropriate. If the same data is required, the data must be copied to each process. The key characteristic is that there is no interaction between the processes.

In a practical embarrassingly parallel computation, data has to be distributed to the processes and results collected and combined in some way. This suggests that initially, and finally, a single process must be operating alone. A common approach is the master-slave organization. If dynamic process creation is used, first, a master process will be started that will spawn (start) identical slave processes. The resulting structure is shown in Figure 3.2. (The master process could take on a computation after spawning, although often this is not done when the master is needed for the results as soon as they arrive.) As noted in Section 2.2.3, the master-slave approach can be used with static process creation. There, we simply put both the master and the slave in the same program and use `IF` statements to select either the master code or the slave code based upon the process identification, ID (the master ID or a slave ID). The actual details of master and slave startup are omitted from the example pseudocode sequences given later.

In this chapter, we consider applications where there is minimal interaction between slave processes. Even if the slave processes are all identical, it may be that statically assigning processes to processors will not provide the optimum solution. This holds especially when the processors are different, as is often the case with networked workstations, and then load-balancing techniques offer improved execution speed. We will introduce load balancing in this chapter, but only for cases in which there is no interaction between slave processes. When there is interaction between processes, load balancing requires a significantly different treatment, and this is addressed in Chapter 7.

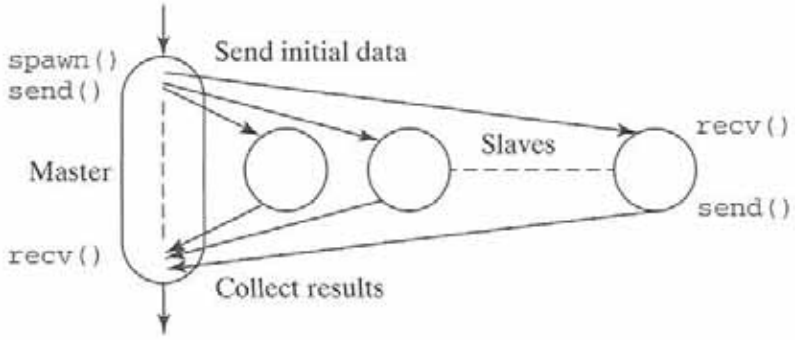


Figure 3.2 Practical embarrassingly parallel computational graph with dynamic process creation and the master-slave approach.

3.2 EMBARRASSINGLY PARALLEL EXAMPLES

3.2.1 Geometrical Transformations of Images

Images are often stored within a computer so that they can be altered in some way. Displayed images originate in two ways. Images are obtained from external sources such as video cameras and may need to be altered in some way (*image processing*). Displayed images may also be artificially created, an approach that is usually associated with the term *computer graphics*. In any event, a number of graphical operations can be performed upon the stored image. For example, we might want to move the image to a different place in the display space, decrease or increase its size, or rotate it in two or three dimensions. Such graphical transformations must be done at high speed to be acceptable to the viewer. Frequently, other image-processing operations, such as smoothing and edge detection, are also done on images, especially externally originated images that are "noisy." and are often embarrassingly parallel. Chapter 11 considers these image-processing operations. Here, we shall consider simple graphical transformations.

The most basic way to store a two-dimensional image is a *pixmap*, in which each pixel (picture element) is stored as a binary number in a two-dimensional array. For purely black-and-white images, a single binary bit is sufficient for each pixel, a 1 if the pixel is white and a 0 if the pixel is black; this is a *bitmap*. *Grayscale* images require more bits, typically using 8 bits to represent 256 different monochrome intensities. Color requires more specification. Usually, the three primary colors, red, green, and blue (RGB), as used in a monitor, are stored as separate 8-bit numbers. Three bytes could be used for each pixel, one byte for red, one for green, and one for blue, 24 bits in all. A standard image file format using this representation is the "tiff" format.

The storage requirements for color images can be reduced by using a look-up table to hold the RGB representation of the specific colors that happen to be used in the image. For example, suppose only 256 different colors are present. A table of 256 24-bit entries could hold the representation of the colors used. Then each pixel in the image would only need to be 8 bits to select the specific color from the look-up table. This method can be used for external image files, in which case the look-up table is held in the file together with the image. The method can also be used for internally generated images to reduce the size of the video memory, though this is becoming less attractive as video memory becomes less expensive. For this section, let us assume a simple grayscale image. (Color images can be reduced to grayscale images, and this is often done for image processing; see Chapter 11.) The terms *bitmap* and *bit-mapped* are used very loosely for images stored in binary as an array of pixels.

Geometrical transformations require mathematical operations to be performed on the coordinates of each pixel to move the position of the pixel without affecting its value. Since the transformation on each pixel is totally independent from the transformations on other pixels, we have a truly embarrassingly parallel computation. The result of a transformation is simply an updated bitmap. A sample of some common geometrical transformations is given here (Wilkinson and Horrocks, 1987):

(a) Shifting

The coordinates of a two-dimensional object shifted by Δx in the x-dimension and Δy

in the y-dimension are given by

$$x' = x + \Delta x$$
$$y' = y + \Delta y$$

where x and y are the original and x' and y' are the new coordinates.

(b) Scaling

The coordinates of an object scaled by a factor S_x in the x-direction and S_y in the y-direction are given by

$$x' = xS_x$$
$$y' = yS_y$$

The object is enlarged in size when S_x and S_y are greater than 1 and reduced in size when S_x and S_y are between 0 and 1. Note that the magnification or reduction does not need to be the same in both x- and y-directions.

(c) Rotation

The coordinates of an object rotated through an angle θ about the origin of the coordinate system are given by

$$x' = x\cos\theta + y\sin\theta$$
$$y' = -x\sin\theta + y\cos\theta$$

(d) Clipping

This transformation applies defined rectangular boundaries to a figure and deletes from the displayed picture those points outside the defined area. This may be useful after rotation, shifting, and scaling have been applied to eliminate coordinates outside the field of view of the display. If the lowest values of x, y in the area to be displayed are x_l, y_l, and the highest values of x, y are x_h, y_h, then

$$x_l \le x' \le x_h$$
$$y_l \le y' \le y_h$$

need to be true for the point (x', y') to be displayed; otherwise the point (x', y') is not displayed.

The input data is the bitmap that is typically held in a file and copied into an array. The contents of this array can easily be manipulated without any special programming techniques. The main parallel programming concern is the division of the bitmap into groups of pixels for each processor because there are usually many more pixels than processes/processors. There are two general methods of grouping: by square/rectangular regions and by columns/rows. We can simply assign one process(or) to one area of the display. For example, with a 640×480 image and 48 processes, we could divide the display area into 48 80×80 rectangular areas and assign one process for each 80×80 rectangular area. Alternatively, we might divide the area into 48 rows of 640×10 pixels for each process. The concept of dividing an area into either rectangular/square areas of rows (or columns), as shown in Figure 3.3, appears in many applications involving processing two-dimensional

information. We explore the trade-offs between dividing a region into square blocks or rows (or columns) in Chapter 6. For the case where there is no communication between adjacent areas, as here, it does not matter which partitioning we use, except perhaps for ease of programming.

Suppose we use a master process and 48 slave processes and partition in groups of 10 rows. Each slave process processes one 640×10 area, returning the new coordinates to the master for displaying. If the transformation is shifting, as described in (a) previously, a master-slave approach could start with the master sending the first row number of the 10 rows to be processed by each process. Upon receiving its row number, each process steps through each pixel coordinate in its group of rows, transforming the coordinates and sending the old and new coordinates back to the master. For simplicity, this could be done with individual messages rather than a single message. The master then updates the bitmap.

Let the original bitmap be held in the array `map[][]`. A temporary bitmap is declared, `temp_map[][]`. Usually, the coordinate system of the display has its origin at the top left corner, as shown in Figure 3.3. It is simple matter to transform an image with its origin at

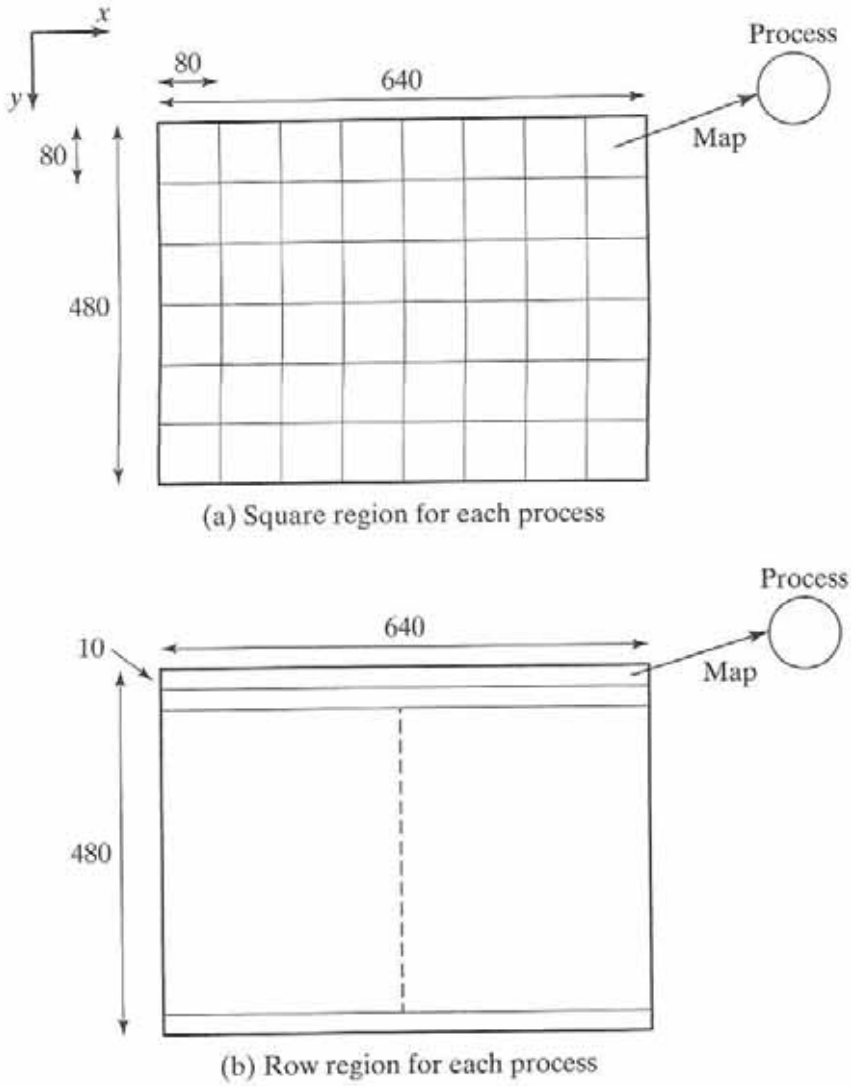


Figure 3.3 Partitioning into regions for individual processes.

the lower-left corner to the display coordinate system. Such details are omitted. Note that in the C programming language, elements are stored row by row, with the first index being the row and the second index being the column. The pseudocode to perform an image shift might look like this:

Master

```
for (i = 0, row = 0; i < 48; i++, row = row + 10)   /* for each process*/
   send(row, Pi);                                    /* send row no.*/

for (i = 0; i < 480; i++)                            /* initialize temp */
   for (j = 0; j < 640; j++)
      temp_map[i][j] = 0;

for (i = 0; i < (640 * 480); i++) {                  /* for each pixel */
   recv(oldrow,oldcol,newrow,newcol, PANY);          /* accept new coords */
   if !((newrow < 0)||(newrow >= 480)||(newcol < 0)||(newcol >= 640))
      temp_map[newrow][newcol]=map[oldrow][oldcol];
}
for (i = 0; i < 480; i++)                            /* update bitmap */
   for (j = 0; j < 640; j++)
      map[i][j] = temp_map[i][j];
```

Slave

```
recv(row, Pmaster);                                 /* receive row no. */
for (oldrow = row; oldrow < (row + 10); oldrow++)
   for (oldcol = 0; oldcol < 640; oldcol++) {       /* transform coords */
      newrow = oldrow + delta_x;                    /* shift in x direction */
      newcol = oldcol + delta_y;                    /* shift in y direction */
      send(oldrow,oldcol,newrow,newcol, Pmaster);   /* coords to master */
   }
```

In the master's receive section, we show a wildcard, P_{any}, indicating that data may be accepted from any slave and in any order. It also may be possible for images to be shifted past the edges of the display area. Wherever the new image does not appear in the display area, the bitmap is set to 0 (black).

In this example, the master sends the starting row numbers to the slaves. However, since the starting numbers are related to the process ID, each slave can determine its starting row itself. The results are returned one at a time rather than as one group, which would have reduced the message overhead time. No code is shown for terminating the slaves, and all results must be generated or the master will wait forever. Slaves here can terminate themselves after they have completed their fixed number of tasks; in other cases, they might be terminated by the master sending them a message to do so. The code is not completely satisfactory because the size of the display area, number of rows for each process, and number of processes are permanently coded for ease of discussion. For complete generality, these factors should be made easily alterable.

Analysis. Suppose each pixel requires two computational steps and there are $n \times n$ pixels. If the transformations are done sequentially, there would be $n \times n$ steps so that

$$t_s = 2n^2$$

and a sequential time complexity of $O(n^2)$.

The parallel time complexity is composed of two parts, communication and computation, as are all message-passing parallel computations. Throughout the text, we will deal with communication and computation separately and sum the components of each to form the overall parallel time complexity.

Communication. To recall our basis for communication analysis, as given in Section 2.3.2, we assume that processors are directly connected and start with the formula

$$t_{\text{comm}} = t_{\text{startup}} + mt_{\text{data}}$$

where t_{startup} is the (constant) time to form the message and initiate the transfer, t_{data} is the (constant) time to send one data item, and there are m data items. The parallel time complexity of the communication time, as given by t_{comm}, is $O(m)$. However, in practice, we usually cannot ignore the communication startup time, t_{startup}, because the startup time is significant unless m was really large.

Let the number of processes be p. Before the computation, the starting row number must be sent to each process. In our pseudocode, we send the row numbers sequentially; that is, p `send()`s, each with one data item. The individual processes have to send back the transformed coordinates of their group of pixels, shown here with individual `send()`s. There are $4n^2$ data items returned to the master, which it will have to accept sequentially. Hence, the communication time is

$$t_{\text{comm}} = p(t_{\text{startup}} + t_{\text{data}}) + n^2(t_{\text{startup}} + 4t_{\text{data}}) = O(p + n^2)$$

Computation. The parallel implementation (using groups of rows or columns or square/rectangular regions) divides the image into groups of n^2/p pixels. Each pixel requires two additions (see slave pseudocode on page 84). Hence, the parallel computation time is given by

$$t_{\text{comp}} = 2\left(\frac{n^2}{p}\right) = O(n^2/p)$$

Overall execution time. The overall execution time is given by

$$t_p = t_{\text{comp}} + t_{\text{comm}}$$

For a fixed number of processors, this is $O(n^2)$.

Speedup factor. The speedup factor is

$$\text{Speedup factor} = \frac{t_s}{t_p} = \frac{2n^2}{2\left(\frac{n^2}{p}\right) + p(t_{\text{startup}} + t_{\text{data}}) + n^2(t_{\text{startup}} + 4t_{\text{data}})}$$

Computation/communication ratio. As introduced in Chapter 1, the ratio of the computation and communication times:

$$\text{Computation/communication ratio} = \frac{\text{Computation time}}{\text{Communication time}} = \frac{t_{\text{comp}}}{t_{\text{comm}}}$$

enables one to see the effects of the communication overhead, especially for increasing problem size. In the problem under consideration, the ratio is

$$\text{Computation/communication ratio} = \frac{2(n^2/p)}{p(t_{\text{startup}} + 2t_{\text{data}}) + 4n^2(t_{\text{startup}} + t_{\text{data}})}$$

$$= O\left(\frac{n^2/p}{p + n^2}\right)$$

which is constant as the problem size grows with a fixed number of processors. This is not good for a computation/communication ratio! The ideal time complexity of sequential algorithms is one of smallest order (least growth). Conversely, the ideal computation/communication ratio is one of largest order, as then increasing the problem size reduces the effect of the communication, which is usually very significant. (One could use the communication/computation ratio and then the most desirable ratio will be of smallest order as sequential time complexity.)

In fact, the constant hidden in the communication part far exceeds the constants in the computation in most practical situations. Here, we have $4n^2 + p$ startup times in t_{comm}. This code will perform badly. It is very important to reduce the number of messages. We could broadcast the set of row numbers to all processes to reduce the effects of the startup time. Also, we could send the results back in groups. Even then, the communication dominates the overall execution time since the computation is minimal. This application is probably best suited for a shared memory multiprocessor in which the bitmap would be stored in the shared memory, which would be immediately available to all processors.

3.2.2 Mandelbrot Set

Displaying the Mandelbrot set is another example of processing a bit-mapped image. However, now the image must be computed, and this involves significant computation. A Mandelbrot set is a set of points in a complex plane that are quasi-stable (will increase and decrease, but not exceed some limit) when computed by iterating a function, usually the function

$$z_{k+1} = z_k^2 + c$$

where z_{k+1} is the $(k + 1)$th iteration of the complex number $z = a + bi$ (where $i = \sqrt{-1}$), z_k is the kth iteration of z, and c is a complex number giving the position of the point in the complex plane. The initial value for z is zero. The iterations are continued until the magnitude of z is greater than 2 (which indicates that z will eventually become infinite) or the number of iterations reaches some arbitrary limit. The magnitude of z is the length of the vector given by

$$z_{\text{length}} = \sqrt{a^2 + b^2}$$

Computing the complex function, $z_{k+1} = z_k^2 + c$, is simplified by recognizing that

$$z^2 = a^2 + 2abi + bi^2 = a^2 - b^2 + 2abi$$

or a real part that is $a^2 - b^2$ and an imaginary part that is $2ab$. Hence, if z_{real} is the real part of z, and z_{imag} is the imaginary part of z, the next iteration values can be produced by computing:

$$z_{real} = z_{real}^2 - z_{imag}^2 + c_{real}$$

$$z_{imag} = 2z_{real}z_{imag} + c_{imag}$$

where c_{real} is the real part of c, and c_{imag} is the imaginary part of c.

Sequential Code. For coding, a structure can be used holding both the real and imaginary parts of z:

```
structure complex {
   float real;
   float imag;
};
```

A routine for computing the value of one point and returning the number of iterations could be of the form

```
int cal_pixel(complex c)
{
int count, max_iter;
complex z;
float temp, lengthsq;
max_iter = 256;
z.real = 0;
z.imag = 0;
count = 0;                                  /* number of iterations */
do {
   temp = z.real * z.real - z.imag * z.imag + c.real;
   z.imag = 2 * z.real * z.imag + c.imag;
   z.real = temp;
   lengthsq = z.real * z.real + z.imag * z.imag;
   count++;
} while ((lengthsq < 4.0) && (count < max_iter));
return count;
}
```

The square of the length, `lengthsq`, is compared against 4, rather than the length against 2, to avoid a square root operation. Given the termination conditions, all the Mandelbrot points must be within a circle with its center at the origin and of radius 2.

The code for computing and displaying the points requires some scaling of the coordinate system to match the coordinate system of the display area. The actual viewing area will usually be a rectangular window of any size and sited anywhere of interest in the complex plane. The resolution is expanded at will to obtain fascinating images. Suppose

the display height is `disp_height`, the display width is `disp_width`, and the point in this display area is (`x`, `y`). If this window is to display the complex plane with minimum values of (`real_min`, `imag_min`) and maximum values of (`real_max`, `imag_max`), each (`x`, `y`) point needs to be scaled by the factors

```
c.real = real_min + x * (real_max - real_min)/disp_width;
c.imag = imag_min + y * (imag_max - imag_min)/disp_height;
```

to obtain the actual complex plane coordinates. For computational efficiency, let

```
scale_real = (real_max - real_min)/disp_width;
scale_imag = (imag_max - imag_min)/disp_height;
```

Including scaling, the code could be of the form

```
for (x = 0; x < disp_width; x++)          /* screen coordinates x and y */
   for (y = 0; y < disp_height; y++) {
      c.real = real_min + ((float) x * scale_real);
      c.imag = imag_min + ((float) y * scale_imag);
      color = cal_pixel(c);
      display(x, y, color);
   }
```

where `display()` is a routine suitably written to display the pixel (x, y) at the computed color (taking into account the position of origin in the display window, if necessary). Typical results are shown in Figure 3.4. A sequential version of the program to generate

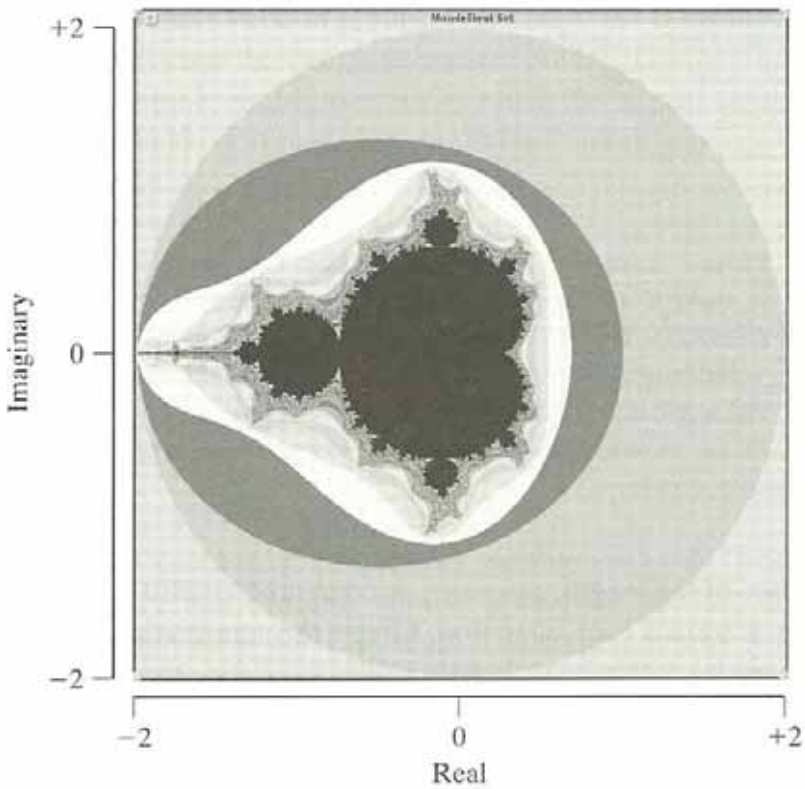


Figure 3.4 Mandelbrot set.

the Mandelbrot set using Xlib calls for the graphics is available at http://www.cs.uncc.edu/par_prog and could be the basis of a simple parallel program (Problem 3-7).

The Mandelbrot set is a widely used test in parallel computer systems (and, for that matter, sequential computers) because it is computationally intensive. It can take several minutes to compute the complete image, depending upon the speed of the computer and the required resolution of the image. Also, it has interesting graphical results. A selected area can be magnified by repeating the computation over that area and scaling the results for the display routine.

Parallelizing the Mandelbrot Set Computation. The Mandelbrot set is particularly convenient to parallelize for a message-passing system because each pixel can be computed without any information about the surrounding pixels. The computation of each pixel itself is less amenable to parallelization. In the previous example of transforming an image, we assigned a fixed area of the display to each process. This is known as *static assignment*. Here, we will consider both static assignment and *dynamic assignment* when the areas computed by a process vary.

Static Task Assignment. Grouping by square/rectangular regions or by columns/rows, as shown in Figure 3.3, is suitable. Each process needs to execute the procedure, `cal_pixel()`, after being given the coordinates of the pixels to compute. Suppose the display area is 640×480 and we were to compute 10 rows in a process (i.e., a grouping of 10×640 pixels and 48 processes). The pseudocode might look like the following:

```
Master

for (i = 0, row = 0; i < 48; i++, row = row + 10)/* for each process*/
   send(&row, P_i);                                /* send row no.*/
for (i = 0; i < (480 * 640); i++) {   /* from processes, any order */
   recv(&c, &color, P_ANY);           /* receive coordinates/colors */
   display(c, color);                 /* display pixel on screen */
}

Slave (process i)

recv(&row, P_master);                 /* receive row no. */
for (x = 0; x < disp_width; x++)      /* screen coordinates x and y */
   for (y = row; y < (row + 10); y++) {
      c.real = min_real + ((float) x * scale_real);
      c.imag = min_imag + ((float) y * scale_imag);
      color = cal_pixel(c);
      send(&c, &color, P_master);     /* send coords, color to master */
   }
```

We expect all 640×480 pixels to be sent, but this may occur in any order dependent upon the number of iterations to compute the pixel value and the speed of the computer. The implementation has the same serious disadvantage as that for transformations in Section 3.2.1; it sends results back one at a time rather than in groups. Sending the data in groups

reduces the number of communication startup times (one for each message). It is a simple matter to save results in an array and then send the whole array to the master in one message. Note that the master uses a wild card to accept messages from slaves in any order (the notation P_{ANY} indicates a source wild card).

Dynamic Task Assignment—Work Pool/Processor Farms. The Mandelbrot set requires significant iterative computation for each pixel. The number of iterations will generally be different for each pixel. In addition, the computers may be of different types or operate at different speeds. Thus, some processors may complete their assignments before others. Ideally, we want all processors to complete together, achieving a system efficiency of 100 percent, which can be addressed using *load balancing*. This is a complex and extremely important concept in all parallel computations and not just the one we are considering here. Ideally, each processor needs to be given sufficient work to keep it busy for the duration of the overall computation. Regions of different sizes could be assigned to different processors, but this would not be very satisfactory for two reasons: We may not know a priori each processor's computational speed, and we would have to know the exact time it takes for each processor to compute each pixel. The latter depends upon the number of iterations, which differs from one pixel to the next. Some problems may be more uniform in their computational time, but, in any event, a more system-efficient approach will involve some form of dynamic load balancing.

Dynamic load balancing can be achieved using a *work-pool* approach, in which individual processors are supplied with work when they become idle. Sometimes the term *processor farm* is used to describe this approach, especially when all the processors are of the same type. The work pool holds a collection, or pool, of tasks to be performed. In some work-pool problems, processes can generate new tasks; we shall look at the implications of this in Chapter 7.

In our problem, the set of pixels (or, more accurately, their coordinates) forms the tasks. The number of tasks is fixed in that the number of pixels to compute is known prior to the start of the computation. Individual processors request a pair of pixel coordinates from the work pool. When a processor has computed the color for the pixel, it returns the color and requests a further pair of pixel coordinates from the work pool. When all the pixel coordinates have been taken, we then need to wait for all the processors to complete their tasks and report in for more pixel coordinates.

Sending pairs of coordinates of individual pixels will result in excessive communication. Normally, rather than a single coordinate comprising a task, a group of coordinates representing several pixels will form the task to be taken from the work pool in order to reduce the communication overhead. At the start, the slaves are told the size of the group of pixels (assumed to be a fixed size). Then only the first pair of coordinates of the group need be sent to the slaves as a task. This approach reduces the communication to an acceptable level. The overall arrangement is depicted in Figure 3.5.

When the Mandelbrot-set calculation is coded for a work-pool solution, we will find that the pixels will not be generated together. Some will appear before others. (Actually, this will also happen in our example code for the static assignment because some pixels will require more time than others and the order in which messages are received is not constrained.)

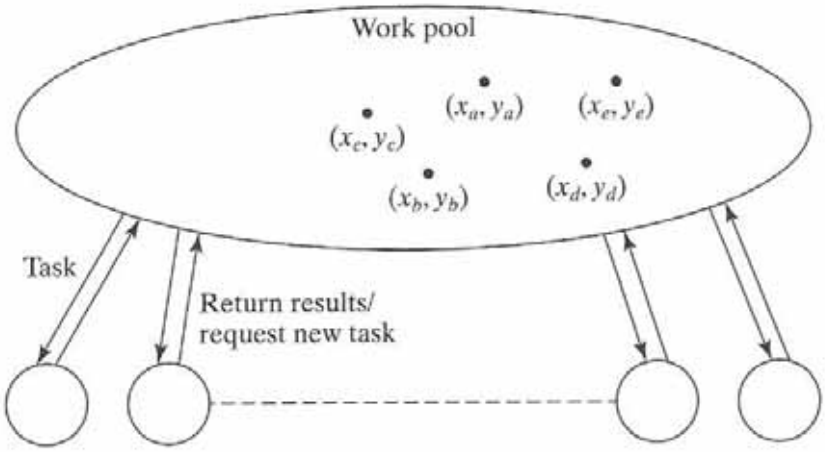


Figure 3.5 Work-pool approach.

Suppose the number of processes is given by num_proc, and processes compute one row at a time. In this case, the work pool holds row numbers rather than individual pixel coordinates. Coding for a work pool could be of the form

Master

```
count = 0;                                  /* counter for termination*/
row = 0;                                    /* row being sent */
for (k = 0; k < num_proc; k++) {          /* assuming num_proc<disp_height */
   send(&row, Pk, data_tag);                /* send initial row to process */
   count++;                                 /* count rows sent */
   row++;                                   /* next row */
}

do {
   recv (&slave, &r, color, PANY, result_tag);
   count--;                              /* reduce count as rows received */
   if (row < disp_height) {
      send (&row, Pslave, data_tag);                    /* send next row */
      row++;                                            /* next row */
      count++;
   } else
      send (&row, Pslave, terminator_tag);              /* terminate */
   display (r, color);                                  /* display row */
} while (count > 0);
```

Slave

```
recv(y, Pmaster, ANYTAG, source_tag);     /* receive 1st row to compute */
while (source_tag == data_tag) {
   c.imag = imag_min + ((float) y * scale_imag);
   for (x = 0; x < disp_width; x++) {              /* compute row colors */
      c.real = real_min + ((float) x * scale_real);
      color[x] = cal_pixel(c);
   }
```

```
    send(&x, &y, color, P_master, result_tag);        /* row colors to master */
    recv(&y, P_master, source_tag);                   /* receive next row */
};
```

In this code, each slave is first given one row to compute and, from then on, gets another row when it returns a result, until there are no more rows to compute. The master sends a terminator message when all the rows have been taken. To differentiate between different messages, tags are used with the message `data_tag` for rows being sent to the slaves, `terminator_tag` for the terminator message, and `result_tag` for the computed results from the slaves. It is then necessary to have a mechanism to recognize different tags being received. Here we simply show a `source_tag` parameter. Note that the master receives and sends messages before displaying results, which can allow the slaves to restart as soon as possible. Locally blocking sends are used. Note also that in order to terminate, the number of rows outstanding in the slaves is counted (`count`), as illustrated in Figure 3.6. It is also possible simply to count the number of rows returned. There are, of course, other ways to code this problem. Termination is considered further in Chapter 7.

Analysis. Exact analysis of the Mandelbrot computation is complicated by not knowing how many iterations are needed for each pixel. Suppose there are n pixels. The number of iterations for each pixel is some function of `c` but cannot exceed `max_iter`. Therefore, the sequential time is

$$t_s \leq \texttt{max_iter} \times n$$

or a sequential time complexity of $O(n)$.

For the parallel version, let us just consider the static assignment. The total number of processors is given by p, and there are $p - 1$ slave processors. The parallel program has essentially three phases: communication, computation, and communication.

Phase 1: Communication. First, the row number is sent to each slave, one data item to each of $p - 1$ slaves:

$$t_{\text{comm1}} = (p - 1)(t_{\text{startup}} + t_{\text{data}})$$

Separate messages are sent to each slave, causing duplicated startup times. It would be possible to use a scatter routine, which should reduce this effect (Problem 3-6).

Phase 2: Computation. Then the slaves perform their Mandelbrot computation in parallel:

$$t_{\text{comp}} \leq \frac{\texttt{max_iter} \times n}{p - 1}$$

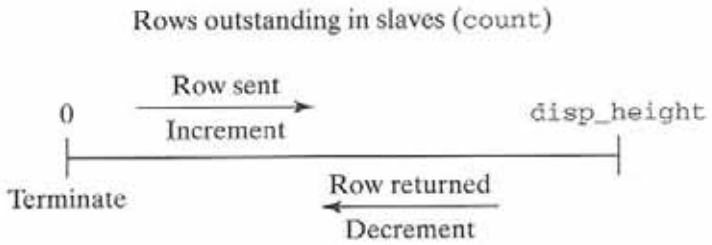


Figure 3.6 Counter termination.

assuming the pixels are evenly divided across all the processors. At least some of the pixels will require the maximum number of iterations, and these will dominate the overall time.

Phase 3: Communication. In the final phase, the results are passed back to the master, one row of pixel colors at a time. Suppose each slave handles u rows and there are v pixels on a row. Then:

$$t_{\text{comm2}} = u(t_{\text{startup}} + vt_{\text{data}})$$

The startup time overhead could be reduced by collecting results into fewer messages. For static assignment, both the value for v (pixels on a row) and value for u (number of rows) would be fixed (unless the resolution of the image was changed). Let us assume that $t_{\text{comm2}} = k$, a constant.

Overall Execution Time. Overall, the parallel time is given by

$$t_p \leq \frac{\texttt{max_iter} \times n}{p-1} + (p-1)(t_{\text{startup}} + t_{\text{data}}) + k$$

Speedup Factor. The speedup factor is

$$\text{Speedup factor} = \frac{t_s}{t_p} = \frac{\texttt{max_iter} \times n}{\dfrac{\texttt{max_iter} \times n}{p-1} + (p-1)(t_{\text{startup}} + t_{\text{data}}) + k}$$

The potential speedup approaches p if `max_iter` is large.

Computation/communication Ratio. The computation/communication ratio

$$\text{Computation/communication ratio} = \frac{(\texttt{max_iter} \times n)}{(p-1)((p-1)(t_{\text{startup}} + t_{\text{data}}) + k)}$$

$$= O(n) \text{ with a fixed number of processors}$$

The preceding analysis is only intended to give an indication of whether parallelism is worthwhile. It appears worthwhile.

3.2.3 Monte Carlo Methods

The basis of Monte Carlo methods is the use of random selections in calculations that lead to the solution to numerical and physical problems. Each calculation will be independent of the others and therefore amenable to embarrassingly parallel methods. The name *Monte Carlo* comes from the similarity of statistical simulation methods to the randomness of games of chance; Monte Carlo, the capital of Monaco, is a center for gambling. (The name is attributed to Metropolis on the Manhattan Project during World War II.)

An example that has reappeared in the literature many times (Fox et al., 1988; Gropp, Lusk, and Skjellum, 1999; Kalos and Whitlock, 1986) is to calculate π as follows. A circle is formed within a square, as shown in Figure 3.7. The circle has unit radius, so that the square has sides 2×2. The ratio of the area of the circle to the square is given by

$$\frac{\text{Area of circle}}{\text{Area of square}} = \frac{\pi(1)^2}{2 \times 2} = \frac{\pi}{4}$$

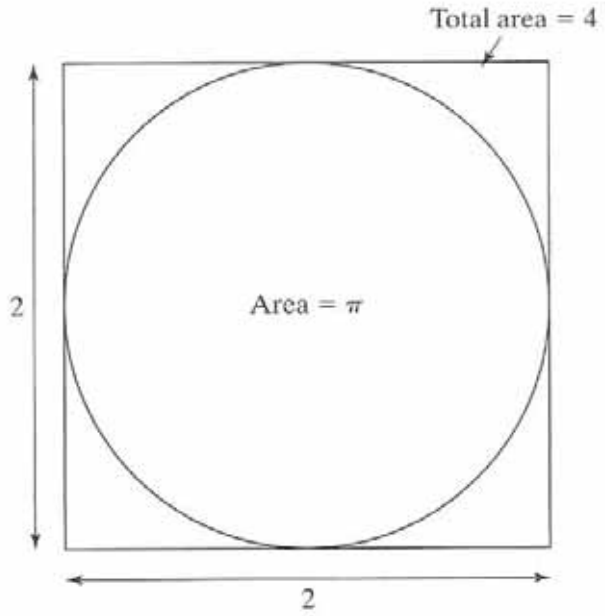


Figure 3.7 Computing π by a Monte Carlo method.

(The same result is obtained for a circle of any dimensions so long as the circle fits exactly inside the square.) Points within the square are chosen randomly and a score is kept of how many points happen to lie within the circle. The fraction of points within the circle will be $\pi/4$, given a sufficient number of randomly selected samples.

The area of any shape within a known bound area could be computed by the preceding method, or any area under a curve; that is, an integral. One quadrant of the construction in Figure 3.7 as a function is illustrated in Figure 3.8, which can be described by the integral

$$\int_0^1 \sqrt{1-x^2}\,dx = \frac{\pi}{4}$$

(positive square roots). A random pair of numbers, (x_r, y_r), would be generated, each between 0 and 1, and then counted as in the circle if $y_r \le \sqrt{1-x_r^2}$; that is, $y_r^2 + x_r^2 \le 1$.

The method could be used to compute any definite integral. Unfortunately, it is very inefficient and also requires the maximum and minimum values of the function within the region of interest. An alternative probabilistic method for finding an integral is to use the random values of x to compute $f(x)$ and sum the values of $f(x)$:

$$\text{Area} = \int_{x_1}^{x_2} f(x)\,dx = \lim_{N\to\infty} \frac{1}{N}\sum_{r=1}^{N} f(x_r)(x_2 - x_1)$$

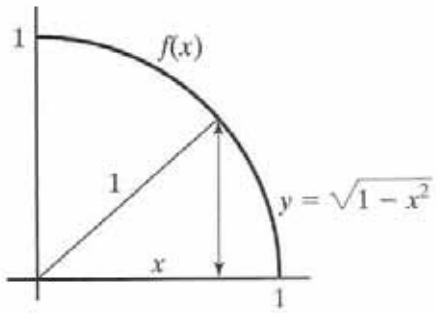


Figure 3.8 Function being integrated in computing π by a Monte Carlo method.

where x_r are randomly generated values of x between x_1 and x_2. This method could also be considered a Monte Carlo method, though some would limit Monte Carlo methods to those that select random values not in the solution as well as those in the solution. The substantial mathematical underpinning to Monte Carlo methods can be found in texts such as Kalos and Whitlock (1986). Monte Carlo methods would not be used in practice for one-dimensional integrals, for which quadrature methods (Section 4.2.2) are better. However, they would be very useful for integrals with a large number of variables and become practical in these circumstances.

Let us briefly look at how a Monte Carlo method could be implemented using $f(x) = x^2 - 3x$ as a concrete example; that is, computing the integral

$$I = \int_{x_1}^{x_2} (x^2 - 3x)\, dx$$

Sequential Code. The sequential code might be of the form

```
sum = 0;
for (i = 0; i < N; i++) {              /* N random samples */
   xr = rand_v(x1, x2);                /* generate next random value */
   sum = sum + xr * xr - 3 * xr;       /* compute f(xr) */
}
area = (sum / N) * (x2 - x1);
```

The routine `rand_v(x1, x2)` returns a pseudorandom number between `x1` and `x2`. A fixed number of samples have been taken; in reality, the integral should be computed to some prescribed precision, which requires a statistical analysis to determine the number of samples required. Texts on Monte Carlo methods, such as Kalos and Whitlock (1986), explore statistical factors in great detail.

Parallel Code. The problem is embarrassingly parallel, since the iterations are independent of each other. The main concern is how to produce the random numbers in such a way that each computation uses a different random number and there is no correlation between the numbers. Standard library pseudorandom-number generators such as `rand()`, could be used (but see later). One approach, as used in Gropp, Lusk, and Skjellum (1999), is to have a separate process responsible for issuing the next random number. This structure is illustrated in Figure 3.9. First, the master process starts the slaves, which request a random number from the random-number process for each of their computations. The slaves form their partial sums, which are returned to the master for final accumulation. If each slave is asked to perform the same number of iterations and the system is homogeneous (identical processors), the slaves should complete more or less together.

The random-number process can only service one slave at a time, and this approach has the significant communication cost of sending individual random numbers to the slaves. Groups of random numbers could be sent to reduce the effects of the startup times. The random-number generator could also be incorporated into the master process, since this process is not otherwise active throughout the computation. All this leads to parallel pseudocode of the following form:

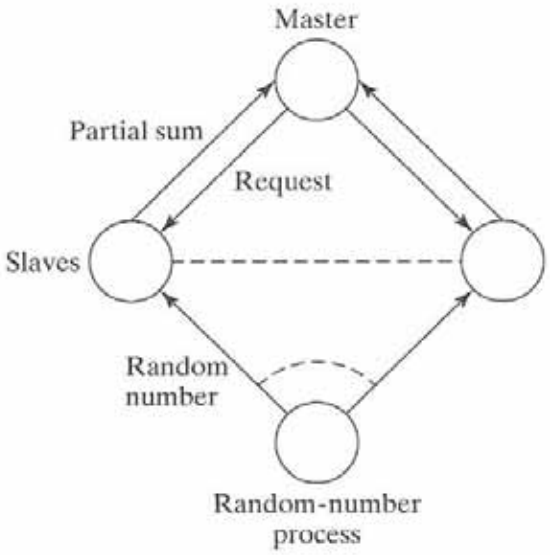


Figure 3.9 Parallel Monte Carlo integration.

Master

```
for(i = 0; i < N/n; i++) {
   for (j = 0; j < n; j++)        /*n=number of random numbers for slave */
      xr[j] = rand();               /* load numbers to be sent */
   recv(P_ANY, req_tag, P_source);     /* wait for a slave to make request */
   send(xr, &n, P_source, compute_tag);
}
for(i = 0; i < num_slaves; i++) {  /* terminate computation */
   recv(P_i, req_tag);
   send(P_i, stop_tag);
}
sum = 0;
reduce_add(&sum, P_group);
```

Slave

```
sum = 0;
send(P_master, req_tag);
recv(xr, &n, P_master, source_tag);
while (source_tag == compute_tag) {
   for (i = 0; i < n; i++)
      sum = sum + xr[i] * xr[i] - 3 * xr[i];
   send(P_master, req_tag);
   recv(xr, &n, P_master, source_tag);
};
reduce_add(&sum, P_group);
```

In this code, the master waits for any slave to respond using a source wild card (P_{ANY}). The rank of the actual slave responding can be obtained from a status call or parameter. We simply show the source within the message envelope. The type of handshaking is reliable but does have more communication than simply sending data without a request; as we have seen, reducing the communication overhead is perhaps the most important aspect for

obtaining high execution speed. It is left as an exercise to eliminate the handshaking. The routine `reduce_add()` is our notation to show a reduce routine that collectively performs addition. The notation to specify a group of processes is P_{group}.

Parallel Random-Number Generation. For successful Monte Carlo simulations, the random numbers must be independent of each other. The most popular way of creating a pseudorandom-number sequence, $x_1, x_2, x_3, \ldots, x_{i-1}, x_i, x_{i+1}, \ldots, x_{n-1}, x_n$, is by evaluating x_{i+1} from a carefully chosen function of x_i. The key is to find a function that will create a very large sequence with the correct statistical properties. The function used is often of the form

$$x_{i+1} = (ax_i + c) \bmod m$$

where a, c, and m are constants chosen to create a sequence that has similar properties to truly random sequences. A generator using this form of equation is known as a *linear congruential generator*. There are many possible values of a, c, and m, and much has been published on the statistical properties of these generators. (See Knuth, 1981, the standard reference, and Anderson, 1990.) A "good" generator is with $a = 16807$, $m = 2^{31} - 1$ (a prime number), and $c = 0$ (Park and Miller, 1988). This generator creates a repeating sequence of $2^{31} - 2$ different numbers (i.e., the maximum for these generators, $m - 1$). A disadvantage of such generators for Monte Carlo simulation is that they are relatively slow.

Even though the pseudorandom-number computation appears to be sequential in nature, each number is calculated from the preceding number and a parallel formulation is possible to increase the speed of generating the sequence. It turns out that

$$x_{i+1} = (ax_i + c) \bmod m$$

$$x_{i+k} = (Ax_i + C) \bmod m$$

where $A = a^k \bmod m$, $C = c(a^{k-1} + a^{k-2} + \ldots + a^1 + a^0) \bmod m$, and k is a selected "jump" constant. Some care is needed to compute and use A and C because of the large numbers involved, but they need be computed only once. (C would not be needed for the good generator described earlier.) Given m processors, the first m numbers of the sequence are generated sequentially. Then each of these numbers can be used to create the next m numbers in parallel, as shown in Figure 3.10, and so on, for the next m numbers.

The computation can be simplified if m is a power of 2, because then the mod operation simply returns the lower m digits. Unfortunately, generators of this type often use a prime number for m to obtain good pseudorandom-number characteristics. Fox, Williams, and Messina (1994) describe a different type of random-number generator using the formula $x_i = (x_{i-63} + x_{i-127}) \bmod 2^{31}$, which naturally generates numbers from distant preceding numbers.

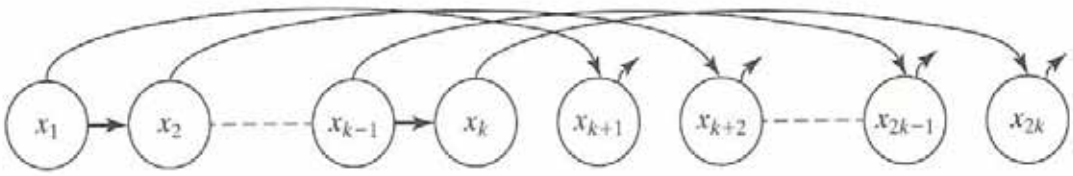


Figure 3.10 Parallel computation of a sequence.

There are several types of pseudorandom-number generators, each using different mathematical formulas. In general, the formula uses previously generated numbers in computing the next number in the sequence, and thus all are deterministic and repeatable. Being repeatable can be advantageous for testing programs, but the formula must produce statistically good random-number sequences. The area of testing random-number generators has a very long history and is beyond the scope of this text. However, one needs to take great care in the choice and use of random-number generators. Some early random-number generators turned out to be quite bad when tested against statistical figures of merit. Even if a random-number generator appears to create a series of random numbers from statistical tests, we cannot assume that different subsequences or samplings of numbers from the sequences are not correlated. This makes using random-number generators in parallel programs fraught with difficulties, because simple naive methods of using them in parallel programs might not produce random sequences and the programmer might not realize this.

In parallel programs in general, one might try to rely on a centralized linear congruential pseudorandom-number generator to send out numbers to slaves processes when they request numbers, as we have described. Apart from incurring a communication overhead and a bottleneck in the centralized communication point, there is a big question mark over whether the subsequence that each slave obtains is truly a random sequence and the sequences themselves are not correlated in some fashion. (One could do statistical tests to confirm this.) Also, all random-number generators repeat their sequences at some point, and using a single random number-generator will make it more likely that the sequence repeats. Alternatively, one might consider a separate pseudorandom-number generator within each slave. However, if the same formula is used even starting at a different number, parts of the same sequence might appear in multiple processors, and even if this does not happen, one cannot immediately assume there is no correlation between sequences. One can see that it is challenging problem to come up with good pseudorandom-sequences for parallel programs.

Because of the importance of pseudorandom-number generators in parallel Monte Carlo computations, effort has gone into finding solid parallel versions of pseudorandom-number generators. SPRNG (Scalable Pseudorandom Number Generator) is a library specifically for parallel Monte Carlo computations and generates random-number streams for parallel processes. This library has several different generators and features for minimization of interprocess correlation and an interface for MPI.

For embarrassingly parallel Monte Carlo simulations where there is absolutely no interaction between concurrent processes, it may be satisfactory to use subsequences which may be correlated if the full sequence is random and the computation is exactly the same as if it were done sequentially. It is left as an exercise to explore this further (Problem 3-14).

3.3 SUMMARY

This chapter introduced the following concepts:

- An (ideal) embarrassingly parallel computation
- Embarrassingly parallel problems and analyses

- Partitioning a two-dimensional data set
- Work-pool approach to achieve load balancing
- Counter termination algorithm
- Monte Carlo methods
- Parallel random-number generation

FURTHER READING

Fox, Williams, and Messina (1994) provide substantial research details on embarrassingly parallel applications (independent parallelism). Extensive details of graphics can be found in Foley et al. (1990). Image processing can be found in Haralick and Shapiro (1992), and we also pursue that topic in Chapter 11. Dewdney (1985) wrote a series of articles on writing programs for computing the Mandelbrot set. Details of Monte Carlo simulations can be found in Halton (1970), Kalos and Whitlock (1986), McCracken (1955), and Smith (1993). Parallel implementations are discussed in Fox et al. (1988) and Gropp, Lusk, and Skjellum (1999). Discussion of parallel random-number generators can be found in Bowan and Robinson (1987), Foster (1995), Fox, Williams, and Messina (1994), Hortensius, McLeod, and Card (1989), and Wilson (1995). Random-number generators must be very carefully used even in sequential programs. A study of different random number generators in this application can be found in Wilkinson (1989). Masuda and Zimmermann (1996) describe a library of random-number generators specifically for parallel computing. Examples are given using MPI. Details of SPRNG can be found at http://sprng.cs.fsu.edu/main.html.

BIBLIOGRAPHY

ANDERSON, S. (1990), "Random Number Generators," *SIAM Review*, Vol. 32, No. 2, pp. 221–251.

BOWAN, K. O., AND M. T. ROBINSON (1987), "Studies of Random Generators for Parallel Processing," *Proc. 2nd Int. Conf. on Hypercube Multiprocessors*, Philadelphia, pp. 445–453.

BRÄUNL, T. (1993), *Parallel Programming:An Introduction*, Prentice Hall, London.

DEWDNEY, A. K. (1985), "Computer Recreations," *Scientific American*, Vol. 253, No. 2 (August), pp. 16–24.

FOLEY, J. D., A. VAN DAM, S. K. FEINER, AND J. F. HUGHES (1990), *Computer Graphics Principles and Practice*, 2nd ed., Addison-Wesley, Reading, MA.

FOSTER, I. (1995), *Designing and Building Parallel Programs*, Addison-Wesley, Reading, MA.

FOX, G., M. JOHNSON, G. LYZENGA, S. OTTO, J. SALMON, AND D. WALKER (1988), *Solving Problems on Concurrent Processors, Vol. 1*, Prentice Hall, Englewood Cliffs, NJ.

FOX, G. C., R. D. WILLIAMS, AND P. C. MESSINA (1994), *Parallel Computing Works*, Morgan Kaufmann, San Francisco, CA.

GROPP, W., E. LUSK, AND A. SKJELLUM (1999), *Using MPI Portable Parallel Programming with the Message-Passing Interfaces*, 2nd edition, MIT Press, Cambridge, MA.

HALTON, J. H. (1970), "A Retrospective and Perspective Survey of the Monte Carlo Method," *SIAM Review*, Vol. 12, No. 1, pp. 1–63.

HARALICK, R. M., AND L. G. SHAPIRO (1992), *Computer and Robot Vision* Volume 1, Addison-Wesley, Reading, MA.

HORTENSIUS, P. D., R. D. MCLEOD, AND H. C. CARD (1989), "Parallel Random Number Generation for VLSI Systems Using Cellular Automata," *IEEE Trans. Comput.*, Vol. 38, No. 10, pp. 1466–1473.

KALOS, M. H., AND P. A. WHITLOCK (1986), *Monte Carlo Methods*, Volume 1, *Basics*, Wiley, NY.

KNUTH, D. (1981), *The Art of Computer Programming*, Volume 2, *Seminumerical Algorithms*, Addison-Wesley, Reading, MA.

MASUDA, N., AND F. ZIMMERMANN (1996), *PRNGlib: A Parallel Random Number Generator Library*, Technical report TR-96-08, Swiss Center for Scientific Computing (available at http://www.cscs.ch/Official/).

MCCRACKEN, D. D. (1955), "The Monte Carlo Method," *Scientific American*, May, pp. 90–96.

PARK, S. K., AND K. W. MILLER (1988), "Random Number Generators: Good Ones Are Hard to Find," *Comm. ACM*, Vol. 31, No. 10, pp. 1192–1201.

SMITH, J. R. (1993), *The Design and Analysis of Parallel Algorithms*, Oxford University Press, Oxford, England.

WILKINSON, B. (1989), "Simulation of Rhombic Cross-Bar Switch Networks for Multiprocessor Systems," *Proc. 20th Annual Pittsburgh Conf. on Modeling and Simulation*, Pittsburgh, May 4–5, pp. 1213–1218.

WILKINSON B., AND D. HORROCKS (1987), *Computer Peripherals*, 2nd edition, Hodder and Stoughton, London.

WILSON, G. V. (1995), *Practical Parallel Programming*, MIT Press, Cambridge, MA.

PROBLEMS

Scientific/Numerical

3-1. Write a parallel program that reads an image file in a suitable uncompressed format (e.g., the PPM format) and generates a file of the image shifted N pixels right, where N is an input parameter.

3-2. Implement the image transformations described in this chapter.

3-3. Rewrite the pseudocode described in Section 3.2.1 to operate on 80×80 square regions rather than groups of rows.

3-4. The windowing transformation involves selecting a rectangular region of interest in an undisplayed picture and transplanting the view obtained onto the display in a specific position. Consider that a rectangular area is selected measuring ΔX by ΔY, with the lower left-hand corner having the coordinates (X, Y) in the undisplayed picture coordinate system. The points within this rectangle, (x, y), are transformed into a rectangle measuring $\Delta X'$ by $\Delta Y'$ by the transformation

$$x' = (\Delta X'/\Delta X)(x - X) + X'$$

$$y' = (\Delta Y'/\Delta Y)(y - Y) + Y'$$

Scaling is involved if $\Delta X'$ is not equal to ΔX and ΔY is not equal to ΔY. Performing the windowing transformation before other transformations, where possible, may reduce the amount of computation on the subsequent transformations. Write a program to perform the windowing transformation.

3-5. A three-dimensional drawing, represented with coordinates of the form (x, y, z), can be projected onto a two-dimensional surface by a perspective transformation. While doing this, hidden lines need to be removed. Beforehand, three-dimensional shifting, scaling, and rotation transformations can be performed. Rotating a three-dimensional object θ degrees about the x-axis requires the transformation

$$x' = x$$
$$y' = y \cos\theta + z \sin\theta$$
$$z' = z \cos\theta - y \sin\theta$$

Similar transformations give rotation about the y- and z-axes. Write a parallel program to perform three-dimensional transformations.

3-6. Rewrite parallel pseudocode for the Mandelbrot computation in Section 3.2.2 using a scatter routine in place of the individual sends for passing the starting rows to each slave. Use a single message in each slave to return its collected results. Analyze your code.

3-7. Download the sequential Mandelbrot program from http/www.cs.uncc.edu/par_prog/ and follow the instructions to compile and run it. (This program uses Xlib calls for the graphics. It is necessary to link the appropriate libraries.) Modify the program to operate a parallel program using static load balancing (i.e., simply divide the image into fixed areas). Instrument the code to obtain the parallel execution time when executing on your system.

3-8. Same as Problem 3-7, only use dynamic load balancing instead.

3-9. Continue Problems 3-7 and 3-8 by experimenting with different starting values for z in the Mandelbrot computation.

3-10. Write a sequential program and a parallel program that compute the fractal ("fractional dimension") image based upon the function

$$z_{i+1} = z_i^3 + c$$

and based upon the function (Bräunl, 1993)

$$z_{i+1} = z_i^3 + (c - 1)z_i - c$$

where $z_0 = 0$ and c provide the coordinates of a point of the image as a complex number.

3-11. Compare the two Monte Carlo ways of computing an integral described in Section 3.2.3 empirically. Use the integral

$$\int_0^1 \sqrt{1 - x^2}\, dx$$

which computes $\pi/4$.

3-12. Rewrite the code for Monte Carlo integration given in Section 3.2.3, eliminating the master process having to request for data explicitly. Analyze your solution.

3-13. Read the paper by Hortensius, McLeod, and Card (1989) and develop code for a parallel random-number generator based upon the method it describes.

3-14. Investigate the effects of using potentially correlated subsequences taken from a random-number generator in embarrassingly parallel Monte Carol simulations. Make a literature search on this subject and write a report.

3-15. The collapse of a set of integers is defined as the sum of the integers in the set. Similarly, the collapse of a single integer is defined as the integer that is the sum of its digits. For example, the collapse of the integer 134957 is 29. This can clearly be carried out recursively, until a single digit results: the collapse of 29 is 11, and its collapse is the single digit 2. The ultimate collapse of a set of integers is just their collapse followed by the result being collapsed recursively until only a single digit $\{0, 1, \ldots, 9\}$ remains. Your task is to write a program that will find the ultimate collapse of a one-dimensional array of N integers. Alternative approaches are as follows:

1. Use K computers in parallel, each adding up approximately N/K of the integers and passing its local sum to the master, which then totals the partial sums and forms the ultimate collapse of that integer.
2. Use K computers in parallel, each doing a collapse of its local set of N/K integers and passing the partial result to a master, which then forms the ultimate collapse of the partial collapses.
3. Use K computers in parallel, each doing an ultimate collapse on each one of its local set of N/K integers individually, then adding the local collapsed integers and collapsing the result recursively to obtain a single digit. Each of the K then sends its digit on to the master for final summing and ultimate collapse.
4. Use one computer to process all N integers according to any of the first three approaches.
5. Extra credit: Prove that the first three approaches are equivalent in that they produce the same digit for the ultimate collapse of the set of N integers.

Real Life

3-16. As Kim knew from her English courses, palindromes are phrases that read the same way right-to-left as they do left-to-right, when disregarding capitalization. The problem title, she recalled, was attributed to Napoleon, who was exiled and died on the island of Elba. Being the mathematically minded type, Kim likened that to her hobby of looking for palindromes on her vehicle odometer: 245542 (her old car), or 002200 (her new car).

Now, in her first job after completing her undergraduate program in computer science, Kim is once again working with palindromes. As part of her work, she is exploring alternative security-encoding algorithms based on the premise that the encoding strings are not palindromes. Her first task is to examine a large set of strings to identify those that are palindromes, so they may be deleted from the list of potential encoding strings.

The strings consist of letters, ($a \ldots z, A \ldots Z$), and numbers, ($0 \ldots 9$). Kim has a one-dimensional array of pointers, `mylist[]`, in which each element points to the start of a string. As with all character strings, the sequence of chars terminates with the null character, '`\0`'. Kim's program is to examine each string and print out all string numbers (the subscripts of `mylist[]` identifying the strings) that correspond to palindromes.

3-17. Andy, Tom, Bill, and Fred have spent most of their freshman year playing a simple card game. They deal out a deck of 52 regular playing cards, 13 to each person. The rules are similar to bridge or spades: teams of two players, seated so that each player has a member of the other team to his left and right. All 52 cards are dealt one at a time in a clockwise manner. The dealer leads first; players take turns playing in a clockwise manner, and must follow suit if possible; the highest card played of the suit led wins the four-card trick unless a trump is played, in which case the highest trump wins. Dealer passes to the left after each hand of 13 tricks has been played. The object is to win the most tricks for your team. Trump

is determined by the player who bids the most tricks; that is, who calls out the highest number of tricks he thinks his team can win. Bidding starts with the dealer and moves clockwise until four consecutive players have announced "no further bid." Each successive bid must be at least one higher than its predecessor. If no one makes a bid, the dealer is stuck with a minimum bid of seven.

Lately, however, one of the group has taken a renewed interest in studying. The result is that fewer than four players are available to play some evenings. Tom decided to write a small game-playing program to fill in for the missing player. Fred wants to make it a parallel computing implementation to allow for the possibility that more than one may be missing. Your job is to assist Fred.

3-18. A small company is having difficulty keeping up with the demand for its services: retrieval of data from a huge database. The company used to just hand a clerk the list of items to be retrieved, and he or she would manually look through the files to find them. The company has progressed far beyond that; now it hands a program a list of items, and the program looks through the database and finds them. Lately the list of items has grown so large, and the retrieval process has become so time-consuming, that customers have begun to complain. The company has offered you the job of reimplementing its retrieval process by using multiple machines in parallel and dividing up the list of items to be retrieved among the machines.

Part 1: Identify all the pitfalls or roadblocks facing you in moving the retrieval process to a parallel processing implementation that are not present in the existing serial/single processor one.

Part 2: Identify one or more solutions to each item identified in Part 1.

Part 3: Simulate a composite solution, retrieving (from a large database) all the items on a list, using multiple processors working in parallel.

3-19. Over the past 35 years a series of unmanned radar-mapping missions have produced a very detailed topographic map of the moon's surface. This information has been digitized and is available in a gridlike format known as a Mercator projection. The topographical data for the next unmanned landing area is contained in a 100×100 array of grid points denoting the height above (or below) the average moon surface level for that 10 km $\times$ 10 km region. This particular landing region was chosen for its gradually changing topography; you may assume that linearly interpolating between any two adjacent grid-points will accurately describe the landscape between the grid points.

Upon the rocket landing somewhere within the designated 10 km $\times$ 10 km region, it will discharge a number of autonomous robots. They will conduct a detailed exploration of the region and report their results back to the rocket via a line-of-sight lightwave communications link (flashes of light emitted by the robots and detected by the rocket). Once its exploration is complete, it is essential that a robot be able to find quickly the nearest location from which line-of-sight communication can occur, since it will have only a short battery life.

The rocket designers have assured us that their receiving antenna will be 20 m above the site of the landing; the transmitting antenna on the robot will be 1 m above its site, wherever that may be in the region. Thus, given only the 100×100 array and the grid-point locations of the rocket and a robot, your job is to determine the grid point nearest the robot that will permit line-of-sight communication with the rocket. You may assume that the topographical data array contains only heights in integer values between +100 m and −100 m, and that both the rocket and the robots will be located on grid points when accessing your program.

3-20. You are given a array of 100×10000 floating point values representing data collected in a series of bake-offs making up the final exam at the Nella School for the Culinary Arts. As with all grading systems, this data must be massaged (normalized) prior to actually assigning grades. For each of the 100 students (whose data is in a row of 10,000 values), the following operations must be performed:

INS (initial normalized score)

> The average of the squares of all data values in a given row that are greater than zero but less than 100.

FNS (final normalized score)

> The value computed for this student's INS, compared to all other students' INS scores. The students whose INS scores are in the top 10 percent overall get an FNS of 4.0; those in the next 20 percent get an FNS of 3.0; those in the next 30 percent get an FNS of 2.0; those in the next 20 percent get an FNS of 1.0; the rest get an FNS of 0.0 and have to enter the bake-off again next year as a result.

Your program is to print out the FNS scores two ways:

1. A list of FNS scores, by student (row number, FNS) for all students
2. A list of students, by FNS (FNS, list of all rows [students] with this FNS) for all FNS values

3-21. Recently, there has been somewhat of a public health scare related to the presence of a bacterium, cryptosporidium, in the water supply system of several municipalities. It has come to our attention that a band of literary terrorists is spreading the bacterium by cleverly and secretly embedding it in novels. You have been hired by a major publishing company to search a new novel for the presence of the word *cryptosporidium*. It is known that the terrorists have resorted to insertion of punctuation, capitalization, and spacing to disguise the presence of *cryptosporidium*; finding instances of the word requires more than doing a simple word search of the text. For example, in a highly publicized case, one page ended with the sentence

> "Leaving his faithful companion, Ospor, to guard the hallway, Tom crept slowly down the stairs and entered the darkened crypt"

while the next page began with

"Ospor, I dium, HELP!" cried Tom, as the giant bats he had disturbed flew around his head. Disaster was narrowly averted when a clerk luckily caught what he thought was a typo and changed "I dium" to "I'm dying" just as the book went to press. Since this publisher handles many books, it is essential that each one be scanned as quickly as possible. To accomplish this you have proposed to use many computers in parallel to divide the task into smaller chunks; each computer would search only a portion of a text. If successful, you stand to make a sizeable commission from the sale of networked computers to the publisher.

Alternative approaches are as follows:

1. Divide the text into sections of equal size and assign each section to a single processor. Each processor checks its section and passes information (about whether it found *cryptosporidium* in its section, portions of the bacterium as the first or last characters of its section, or no evidence of the bacterium at all) back to a master, which examines what was passed back to it and reports on the book as a whole.

2. Divide the text into many more small sections than there are processors and use a work-pool approach, in which faster processors effectively do more of the total work, but in essentially the same manner as described in the preceding approach.

3-22. Nanotechnology is the latest hot field. One of its objectives is to utilize massive numbers of tiny devices operating in parallel to solve problems ranging from environmental decontamination (e.g., cleaning up oil spills), to battlefield cleanup, (removing unexploded ordnance or mines), to exploration and analysis of the Martian surface.

As an expert in parallelism, choose one of these application areas of nanotechnology and discuss the requirements for interdevice communications if it is to ensure that fewer than X percent of whatever it is looking for will be missed.

Chapter 4

Partitioning and Divide-and-Conquer Strategies

In this chapter, we explore two of the most fundamental techniques in parallel programming, *partitioning* and *divide and conquer*. The techniques are related. In partitioning, the problem is divided into separate parts and each part is computed separately. Divide and conquer usually applies partitioning in a recursive manner by continually dividing the problem into smaller and smaller parts before solving the smaller parts and combining the results. First, we will review the technique of partitioning. Then we discuss recursive divide-and-conquer methods. Next we outline some typical problems that can be solved with these approaches. As usual, there is a selection of scientific/numerical and real-life problems at the end of the chapter.

4.1 PARTITIONING

4.1.1 Partitioning Strategies

Partitioning divides the problem into parts. It is the basis of all parallel programming, in one form or another. The embarrassingly parallel problems in the last chapter used partitioning without any interaction between the parts. Most partitioning formulations, however, require the results of the parts to be combined to obtain the desired result. Partitioning can be applied to the program data (i.e., to dividing the data and operating upon the divided data concurrently). This is called *data partitioning* or *domain decomposition*. Partitioning can also be applied to the functions of a program (i.e., dividing it into independent functions and executing them concurrently). This is *functional decomposition*. The idea of

performing a task by dividing it into a number of smaller tasks that when completed will complete the overall task is, of course, well known and can be applied in many situations, whether the smaller tasks operate upon parts of the data or are separate concurrent functions. It is much less common to find concurrent functions in a problem, but data partitioning is a main strategy for parallel programming.

To take a really simple data-partitioning example, suppose a sequence of numbers, $x_0 \ldots x_{n-1}$, are to be added. This is a problem recurring in the text to demonstrate a concept; unless there were a huge sequence of numbers, a parallel solution would not be worthwhile. However, the approach can be used for more realistic applications involving complex calculations on large databases.

We might consider dividing the sequence into p parts of n/p numbers each, $(x_0 \ldots x_{(n/p)-1})$, $(x_{n/p} \ldots x_{(2n/p)-1})$, ..., $(x_{(p-1)n/p} \ldots x_{n-1})$, at which point p processors (or processes) can each add one sequence independently to create partial sums. The p partial sums need to be added together to form the final sum. Figure 4.1 shows the arrangement in which a single processor adds the p partial sums. (The final addition could be parallelized using a tree construction, but that will not be done here.) Note that each processor requires access to the numbers it has to accumulate. In a message-passing system, the numbers would need to be passed to the processors individually. (In a shared memory system, each processor could access the numbers it wanted from the shared memory, and in this respect, a shared memory system would clearly be more convenient for this and similar problems.)

The parallel code for this example is straightforward. For a simple master-slave approach, the numbers are sent from the master processor to the slave processors. They add their numbers, operating independently and concurrently, and send the partial sums to the master processor. The master processor adds the partial sums to form the result. Often, we talk of processes rather than processors for code sequences, where one process is best mapped onto one processor.

It is a moot point whether broadcasting the whole list of numbers to every slave or only sending the specific numbers to each slave is best, since in both cases all numbers must be sent from the master. The specifics of the broadcast mechanism would need to be known in order to decide on the relative merits of the mechanism. A broadcast will have a single startup time rather than separate startup times when using multiple send routines and may be preferable.

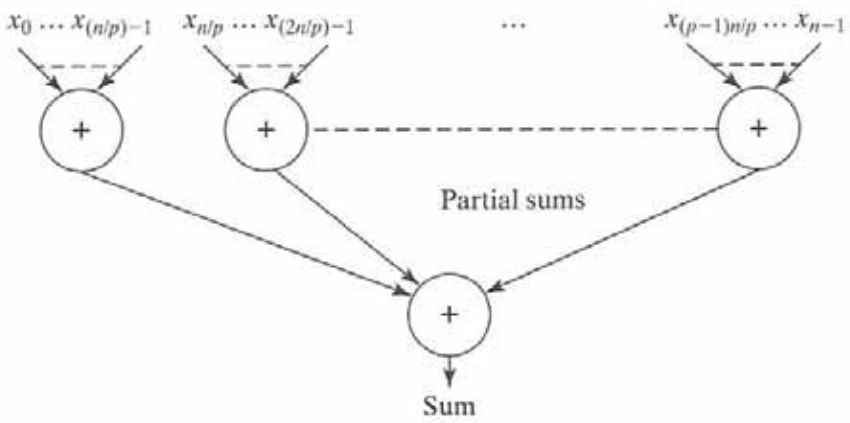


Figure 4.1 Partitioning a sequence of numbers into parts and adding them.

First, we will send the specific numbers to each slave using individual `send()`s. Given n numbers divided into p groups, where n/p is an integer and one group is assigned to one slave process, there would be p slaves. The code using separate `send()`s and `recv()`s might look like the following:

Master

```
s = n/p;                                  /* number of numbers for slaves*/
for (i = 0, x = 0; i < p; i++, x = x + s)
   send(&numbers[x], s, P_i);             /* send s numbers to slave */

sum = 0;
for (i = 0; i < p; i++) {                 /* wait for results from slaves */
   recv(&part_sum, P_ANY);
   sum = sum + part_sum;                  /* accumulate partial sums */
}
```

Slave

```
recv(numbers, s, P_master);               /* receive s numbers from master */
part_sum = 0;
for (i = 0; i < s; i++)                   /* add numbers */
   part_sum = part_sum + numbers[i];
send(&part_sum, P_master);                /* send sum to master */
```

If a broadcast or multicast routine is used to send the complete list to every slave, code is needed in each slave to select the part of the sequence to be used by the slave, adding additional computation steps within the slaves, as in

Master

```
s = n/p;                                  /* number of numbers for slaves */
bcast(numbers, s, P_slave_group);         /* send all numbers to slaves */
sum = 0;
for (i = 0; i < p; i++) {                 /* wait for results from slaves */
   recv(&part_sum, P_ANY);
   sum = sum + part_sum;                  /* accumulate partial sums */
}
```

Slave

```
bcast(numbers, s, P_master);              /* receive all numbers from master*/
start = slave_number * s;                 /* slave number obtained earlier */
end = start + s;
part_sum = 0;
for (i = start; i < end; i++)             /* add numbers */
   part_sum = part_sum + numbers[i];
send(&part_sum, P_master);                /* send sum to master */
```

Slaves are identified by a process ID, which can usually be obtained by calling a library routine. Most often, a group of processes will first be formed and the slave number is an instance or rank within the group. The instance or rank is an integer from 0 to $m - 1$, where there are m processes in the group. MPI requires communicators to be established, and processes have rank within a communicator, as described in Chapter 2. Groups can be associated within communicators, and processes have a rank within groups.

If scatter and reduce routines are available, as in MPI, the code could be

Master

```
s = n/p;                                        /* number of numbers */
scatter(numbers,&s,P_group,root=master);        /* send numbers to slaves */
reduce_add(&sum,&s,P_group,root=master);        /* results from slaves */
```

Slave

```
scatter(numbers,&s,P_group,root=master);        /* receive s numbers */
                                                /* add numbers */
.
.
.
reduce_add(&part_sum,&s,P_group,root=master);   /* send sum to master */
```

Remember, a simple pseudocode is used throughout. Scatter and reduce (and gather when used) have many additional parameters in practice that include both source and destination IDs. Normally, the operation of a reduce routine will be specified as a parameter and not as part of the routine name as here. Using a parameter does allow different operations to be selected easily. Code will also be needed to establish the group of processes participating in the broadcast, scatter, and reduce.

Although we are adding numbers, many other operations could be performed instead. For example, the maximum number of the group could be found and passed back to the master in order for the master to find the maximum number of all those passed back to it. Similarly, the number of occurrences of a number (or character, or string of characters) can be found in groups and passed back to the master.

Analysis. The sequential computation requires $n - 1$ additions with a time complexity of $O(n)$. In the parallel implementation, there are p slaves. For the analysis of the parallel implementation, we shall assume that the operations of the master process are included in one of the slaves in a SPMD model as this is probably done in a real implementation. (Remember that in MPI, data in the root is used in collective operations.) Thus, the number of processors is p. Our analyses throughout separate communication and computation. It is easier to visualize if we also separate the actions into distinct phases. As with many problems, there is a communication phase followed by a computation phase, and these phases are repeated.

Phase 1 — Communication. First, we need to consider the communication aspect of the p slave processes reading their n/p numbers. Using individual send and receive routines requires a communication time of

$$t_{\text{comm1}} = p(t_{\text{startup}} + (n/p)t_{\text{data}})$$

where $t_{startup}$ is the constant time portion of the transmission, and t_{data} is the time to transmit one data word. Using scatter might reduce the number of startup times. Thus,

$$t_{comm1} = t_{startup} + nt_{data}$$

depending upon the implementation of scatter. In any event, the time complexity is still $O(n)$.

Phase 2 — Computation. Next, we need to estimate the number of computational steps. The slave processes each add n/p numbers together, requiring $n/p - 1$ additions. Since all p slave processes are operating together, we can consider all the partial sums obtained in the $n/p - 1$ steps. Hence, the parallel computation time of this phase is

$$t_{comp1} = n/p - 1$$

Phase 3 — Communication. Returning partial results using individual send and receive routines has a communication time of

$$t_{comm2} = p(t_{startup} + t_{data})$$

Using gather and reduce has:

$$t_{comm2} = t_{startup} + pt_{data}$$

Phase 4 — Computation. For the final accumulation, the master has to add the p partial sums, which requires $p - 1$ steps:

$$t_{comp2} = p - 1$$

Overall Execution Time. The overall execution time for the problem (with send and receive) is

$$\begin{aligned} t_p &= (t_{comm1} + t_{comm2}) + (t_{comp1} + t_{comp2}) \\ &= p(t_{startup} + (n/p)t_{data} + p(t_{startup} + t_{data}) + (n/p - 1 + p - 1) \\ &= 2pt_{startup} + (n + p)t_{data} + p + n/p - 2 \end{aligned}$$

or

$$t_p = O(n)$$

for a fixed number of processors. We see that the parallel time complexity is the same as the sequential time complexity of $O(n)$. Of course, if we consider only the computation aspect, the parallel formulation is better than the sequential formulation.

Speedup Factor. The speedup factor is

$$\text{Speedup factor} = \frac{t_s}{t_p} = \frac{n-1}{2pt_{startup} + (n+p)t_{data} + p + n/p - 2}$$

which suggests little speedup for a fixed number of processors.

Computation/communication ratio. The computation/communication ratio is given by

$$\text{Computation/communication ratio} = \frac{t_{comp}}{t_{comm}} = \frac{p + n/p - 2}{2pt_{startup} + (n+p)t_{data}}$$

which again, for a fixed number of processors, does not suggest significant opportunity for improvement.

Ignoring the communication aspect, the speedup factor is given by

$$\text{Speedup factor} = \frac{t_s}{t_p} = \frac{n-1}{n/p + p - 2}$$

The speedup tends to p for large n. However, for smaller n, the speedup will be quite low and worsen for an increasing number of slaves, because the $p-1$ slaves are idle during the fourth phase, forming the final result if one is used for that.

Ideally, we want all the processes to be active all of the time, which cannot be achieved with this formulation of the problem. However, another formulation is helpful and is applicable to a very wide range of problems — namely, the divide-and-conquer approach.

4.1.2 Divide and Conquer

The divide-and-conquer approach is characterized by dividing a problem into subproblems that are of the same form as the larger problem. Further divisions into still smaller subproblems are usually done by recursion, a method well known to sequential programmers. The recursive method will continually divide a problem until the tasks cannot be broken down into smaller parts. Then the very simple tasks are performed and results combined, with the combining continued with larger and larger tasks. JáJá (1992) differentiates between when the main work is in dividing the problem and when it is in combining the results. He categorizes the method as divide and conquer when the main work is combining the results, and as partitioning when the main work is dividing the problem. We will not make this distinction but will use the term *divide and conquer* anytime the partitioning is continued on smaller and smaller problems.

A sequential recursive definition for adding a list of numbers is[1]

```
int add(int *s)                              /* add list of numbers, s */
{
   if (number(s) <= 2) return (n1 + n2);     /* see explanation */
   else {
      Divide (s, s1, s2);        /* divide s into two parts, s1 and s2 */
      part_sum1 = add(s1);       /*recursive calls to add sub lists */
      part_sum2 = add(s2);
      return (part_sum1 + part_sum2);
   }
}
```

As in all recursive definitions, a method must be present to terminate the recursion when the division can go no further. In the code, `number(s)` returns the number of numbers in the list pointed to by `s`. If there are two numbers in the list, they are called `n1` and `n2`. If there is one number in the list, it is called `n1`, and `n2` is zero. If there are no numbers, both `n1` and `n2`

[1] As in all of our pseudocode, implementation details are omitted. For example, the length of a list may need to be passed as an argument.

are zero. Separate `if` statements could be used for each of the cases: 0, 1, or 2 numbers in the list. Each would cause termination of the recursive call.

This method can be used for other global operations on a list, such as finding the maximum number. It can also be used for sorting a list by dividing it into smaller and smaller lists to sort. Mergesort and quicksort sorting algorithms are usually described by such recursive definitions; see Cormen, Leiserson, and Rivest (1990). One would never actually use recursion to add a list of numbers when a simple iterative solution exists, but the following is applicable to any problem that is formulated by a recursive divide-and-conquer method.

When each division creates two parts, a recursive divide-and-conquer formulation forms a binary tree. The tree is traversed downward as calls are made and upward when the calls return (a preorder traversal given the recursive definition). A binary tree construction showing the "divide" part of divide and conquer is shown in Figure 4.2, with the final tasks at the bottom and the root at the top. The root process divides the problem into two parts. These two parts are each divided into two parts, and so on until the leaves are reached. There the basic operations of the problem are performed. This construction can be used in the preceding problem to divide the list of numbers first into two parts, then into four parts, and so on, until each process has one equal part of the whole. After adding pairs at the bottom of the tree, the accumulation occurs in a reverse tree construction.

Figure 4.2 shows a complete binary tree; that is, a perfectly balanced tree with all bottom nodes at the same level. This occurs if the task can be divided into a number of parts that is a power of 2. If not a power of 2, one or more bottom nodes will be at one level higher than the others. For convenience, we will assume that the task can be divided into a number of parts that is a power of 2, unless otherwise stated.

Parallel Implementation. In a sequential implementation, only one node of the tree can be visited at a time. A parallel solution offers the prospect of traversing several parts of the tree simultaneously. Once a division is made into two parts, both parts can be processed simultaneously. Though a recursive parallel solution could be formulated, it is easier to visualize it without recursion. The key is realizing that the construction is a tree.

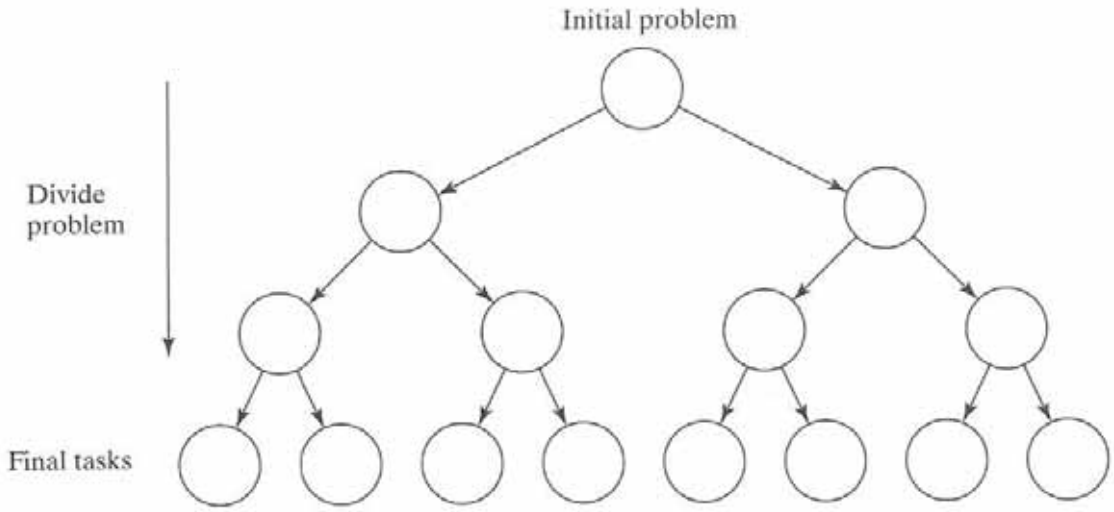


Figure 4.2 Tree construction.

One could simply assign one processor to each node in the tree. That would ultimately require $2^{m+1} - 1$ processors to divide the tasks into 2^m parts. Each processor would only be active at one level in the tree, leading to a very inefficient solution. (Problem 4-5 explores this method.)

A more efficient solution is to reuse processors at each level of the tree, as illustrated in Figure 4.3, which uses eight processors. The division stops when the total number of processors is committed. Until then, at each stage each processor keeps half of the list and passes on the other half. First, P_0 communicates with P_4, passing half of the list to P_4. Then P_0 and P_4 pass half of the list they hold to P_2 and P_6, respectively. Finally, P_0, P_2, P_4, and P_6 pass half of the list they hold to P_1, P_3, P_5, and P_7, respectively. Each list at the final stage will have $n/8$ numbers, or n/p in the general case of p processors. There are $\log p$ levels in the tree.

The "combining" act of summation of the partial sums can be done as shown in Figure 4.4. Once the partial sums have been formed, each odd-numbered processor passes its partial sum to the adjacent even-numbered processor; that is, P_1 passes its sum to P_0, P_3 to P_2, P_5 to P_4, and so on. The even-numbered processors then add the partial sum with its own partial sum and pass the result onward, as shown. This continues until P_0 has the final result.

We can see that these constructions are the same as the binary hypercube broadcast and gather algorithms described in Chapter 2, Section 2.3.3. The constructions would map onto a hypercube perfectly but are also applicable to other systems. As with the hypercube broadcast/gather algorithms, processors that are to communicate with other processors can be found from their binary addresses. Processors communicate with processors whose addresses differ in one bit, starting with the most significant bit for the division phase and with the least significant bit for the combining phase (see Chapter 2, Section 2.3.3).

Suppose we statically create eight processors (or processes) to add a list of numbers. The parallel code for process P_0 might take the form

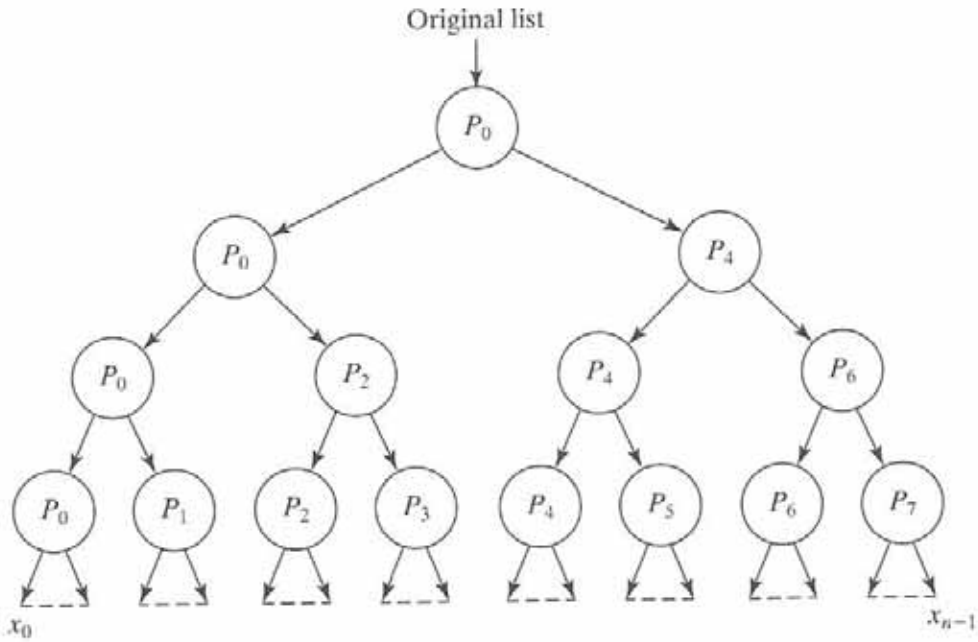


Figure 4.3 Dividing a list into parts.

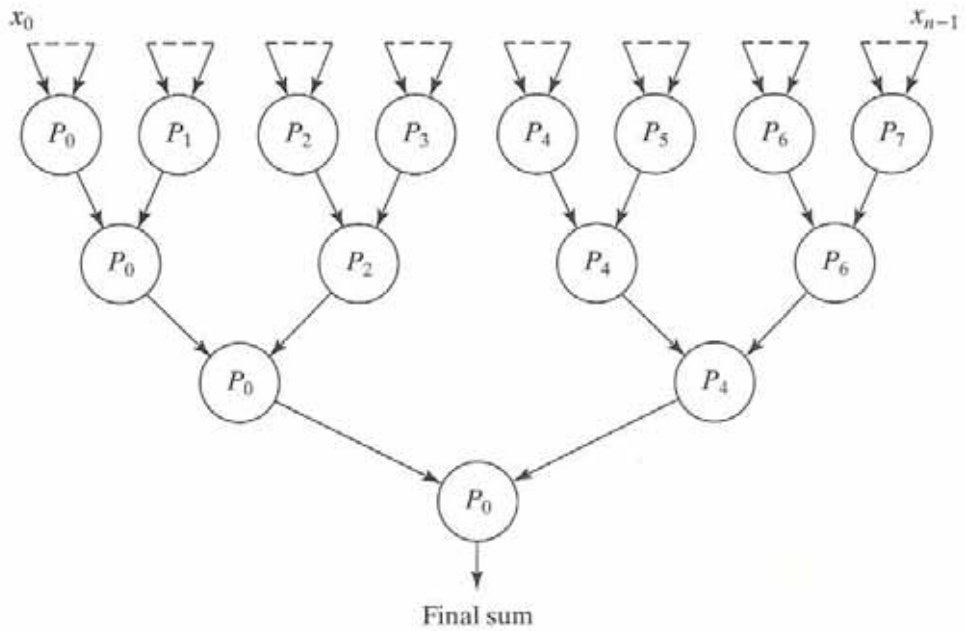


Figure 4.4 Partial summation.

Process P_0

```
                                    /* division phase */
divide(s1, s1, s2);                 /* divide s1 into two, s1 and s2 */
send(s2, P4);                       /* send one part to another process */
divide(s1, s1, s2);
send(s2, P2);
divide(s1, s1, s2);
send(s2, P1);
part_sum = *s1;                     /* combining phase */
recv(&part_sum1, P1);
part_sum = part_sum + part_sum1;
recv(&part_sum1, P2);
part_sum = part_sum + part_sum1;
recv(&part_sum1, P4);
part_sum = part_sum + part_sum1;
```

The code for process P_4 might take the form

Process P_4

```
recv(s1, P0);                       /* division phase */
divide(s1, s1, s2);
send(s2, P6);
divide(s1, s1, s2);
send(s2, P5);
part_sum = *s1;                     /* combining phase */
recv(&part_sum1, P5);
part_sum = part_sum + part_sum1;
recv(&part_sum1, P6);
part_sum = part_sum + part_sum1;
send(&part_sum, P0);
```

Similar sequences are required for the other processes. Clearly, another associative operator, such as multiplication, logical OR, logical AND, minimum, maximum, or string concatenation, can replace the addition operation in the preceding example. The basic idea can also be applied to evaluating arithmetic expressions where operands are connected with an arithmetic operator. The tree construction can also be used for such operations as searching. In this case, the information passed upward is a Boolean flag indicating whether or not the specific item or condition has been found. The operation performed at each node is an OR operation, as shown in Figure 4.5.

Analysis. We shall assume that n is a power of 2. The communication setup time, $t_{startup}$, is not included in the following for simplicity. It is left as an exercise to include the startup time.

The division phase essentially consists only of communication if we assume that dividing the list into two parts requires minimal computation. The combining phase requires both computation and communication to add the partial sums received and pass on the result.

Communication. There is a logarithmic number of steps in the division phase; that is, $\log p$ steps with p processes. The communication time for this phase is given by

$$t_{comm1} = \frac{n}{2}t_{data} + \frac{n}{4}t_{data} + \frac{n}{8}t_{data} + \ldots + \frac{n}{p}t_{data} = \frac{n(p-1)}{p}t_{data}$$

where t_{data} is the transmission time for one data word. The time t_{comm1} is marginally better than a simple broadcast. The combining phase is similar, except that only one data item is sent in each message (the partial sum); that is,

$$t_{comm2} = (\log p)t_{data}$$

for a total communication time of

$$t_{comm} = t_{comm1} + t_{comm2} = \frac{n(p-1)}{p}t_{data} + (\log p)t_{data}$$

or a time complexity of $O(n)$ for a fixed number of processors.

Computation. At the end of the divide phase, the n/p numbers are added together. Then one addition occurs at each stage during the combining phase, leading to

$$t_{comp} = \frac{n}{p} + \log p$$

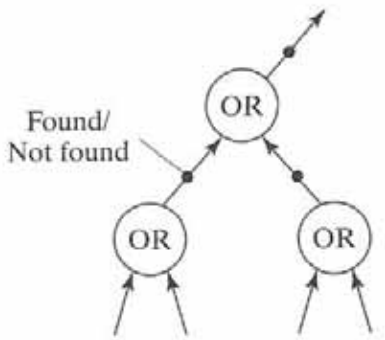


Figure 4.5 Part of a search tree.

again a time complexity of O(n) for a fixed number of processors. For large n and variable p, we get O(n/p).

Overall Execution Time. The total parallel execution time becomes

$$t_p = \left(\frac{n(p-1)}{p} + \log p\right)t_{\text{data}} + \frac{n}{p} + \log p$$

Speedup Factor. The speedup factor is

$$\text{Speedup factor} = \frac{t_s}{t_p} = \frac{n-1}{((n/p)(p-1) + \log p)t_{\text{data}} + n/p + \log p}$$

The very best speedup we could expect with this method is, of course, p when all p processors are computing their partial sums. The actual speedup will be less than this due to the division and combining phases.

Computation/communication Ratio. The computation/communication ratio is given by

$$\text{Computation/communication ratio} = \frac{t_{\text{comp}}}{t_{\text{comm}}} = \frac{n/p + \log p}{((n/p)(p-1) + \log p)t_{\text{data}}}$$

4.1.3 *M*-ary Divide and Conquer

Divide and conquer can also be applied where a task is divided into more than two parts at each stage. For example, if the task is broken into four parts, the sequential recursive definition would be

```
int add(int *s)                          /* add list of numbers, s */
{
   if (number(s) =< 4) return(n1 + n2 + n3 + n4);
   else {
      Divide (s,s1,s2,s3,s4);            /* divide s into s1,s2,s3,s4*/
      part_sum1 = add(s1);               /*recursive calls to add sublists */
      part_sum2 = add(s2);
      part_sum3 = add(s3);
      part_sum4 = add(s4);
      return (part_sum1 + part_sum2 + part_sum3 + part_sum4);
   }
}
```

A tree in which each node has four children, as shown in Figure 4.6, is called a *quadtree*. A quadtree has particular applications in decomposing two-dimensional regions into four subregions. For example, a digitized image could be divided into four quadrants, and then each of the four quadrants divided into four subquadrants, and so on, as shown in Figure 4.7. An *octtree* is a tree in which each node has eight children and has application for dividing a three-dimensional space recursively. An m-ary tree would be formed if the division is into m parts (i.e., a tree with m links from each node), which suggests that greater

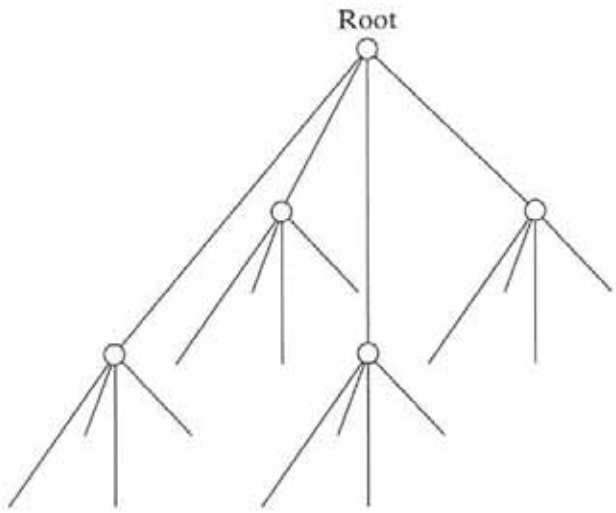


Figure 4.6 Quadtree.

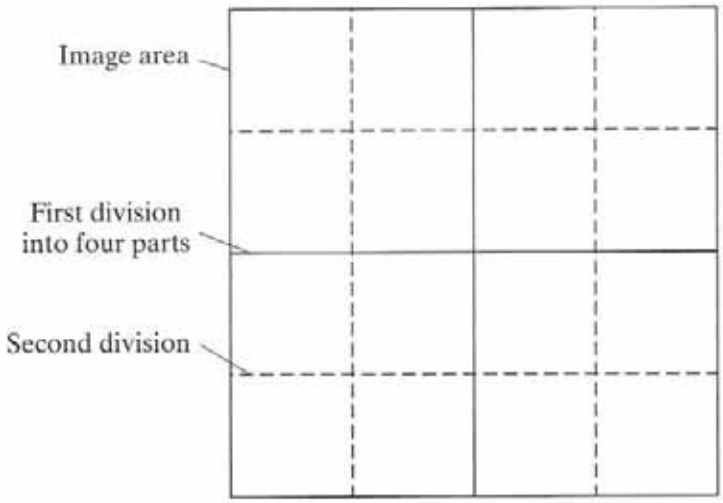


Figure 4.7 Dividing an image.

parallelism is available as m is increased because there are more parts that could be considered simultaneously. It is left as an exercise to develop the equations for computation time and communication time (Problem 4-7).

4.2 PARTITIONING AND DIVIDE-AND-CONQUER EXAMPLES

4.2.1 Sorting Using Bucket Sort

Suppose the problem is not simply to add together numbers in a list, but to sort them into numerical order. There are many practical situations that require numbers to be sorted, and in consequence, sequential programming classes spend a great deal of time developing the various ways that numbers can be sorted. Most of the sequential sorting algorithms are based upon the compare and exchange of pairs of numbers, and we will look at parallelizing such classical sequential sorting algorithms in Chapter 10. Let us look here at the sorting algorithm called *bucket sort*. Bucket sort is not based upon compare and exchange, but is naturally a partitioning method. However, bucket sort only works well if the original numbers are uniformly distributed across a known interval, say 0 to $a - 1$. This interval is

divided into m equal regions, 0 to $a/m - 1$, a/m to $2a/m - 1$, $2a/m$ to $3a/m - 1$, ... and one "bucket" is assigned to hold numbers that fall within each region. There will be m buckets. The numbers are simply placed into the appropriate buckets. The algorithm could be used with one bucket for each number (i.e., $m = n$). Alternatively, the algorithm could be developed into a divide-and-conquer method by continually dividing the buckets into smaller buckets. If the process is continued in this fashion until each bucket can only contain one number, the method is similar to quicksort, except that in quicksort the regions are divided into regions defined by "pivots" (see Chapter 10). Here we will use a limited number of buckets. The numbers in each bucket will be sorted using a sequential sorting algorithm, as shown in Figure 4.8.

Sequential Algorithm. To place a number into a specific bucket it is necessary to identify the region in which the number lies. One way to do this would be to compare the number with the start of regions; i.e., a/m, $2a/m$, $3a/m$, ... This could require as many as $m - 1$ steps for each number on a sequential computer. A more effective way is to divide the number by m/a and use the result to identify the buckets from 0 to $m - 1$, one computational step for each number (although division can be rather expensive in time). If m/a is a power of 2, one can simply look at the upper bits of the number in binary. For example, if $m/a = 2^3$ (eight), and the number is 1100101 in binary, it falls into region 110 (six), by considering the most significant three bits. In any event, let us assume that placing a number into a bucket requires one step, and that placing all the numbers requires n steps. If the numbers are uniformly distributed, there should be n/m numbers in each bucket.

Next, each bucket must be sorted. Sequential sorting algorithms, such as quicksort or mergesort, have a time complexity of $O(n \log n)$ to sort n numbers (average time complexity for quicksort). The lower bound on any compare and exchange sorting algorithm is about $n \log n$ comparisons (Aho, Hopcroft, and Ullman, 1974). Let us assume that the sequential sorting algorithm actually requires $n \log n$ comparisons, one comparison being regarded as one computational step. Thus, it will take $(n/m)\log(n/m)$ steps to sort the n/m numbers in each bucket using these sequential sorting algorithms. The sorted numbers must be concatenated into the final sorted list. Let us assume that this concatenation requires no additional steps. Combining all the actions, the sequential time becomes

$$t_s = n + m((n/m)\log(n/m)) = n + n\log(n/m) = O(n\log(n/m))$$

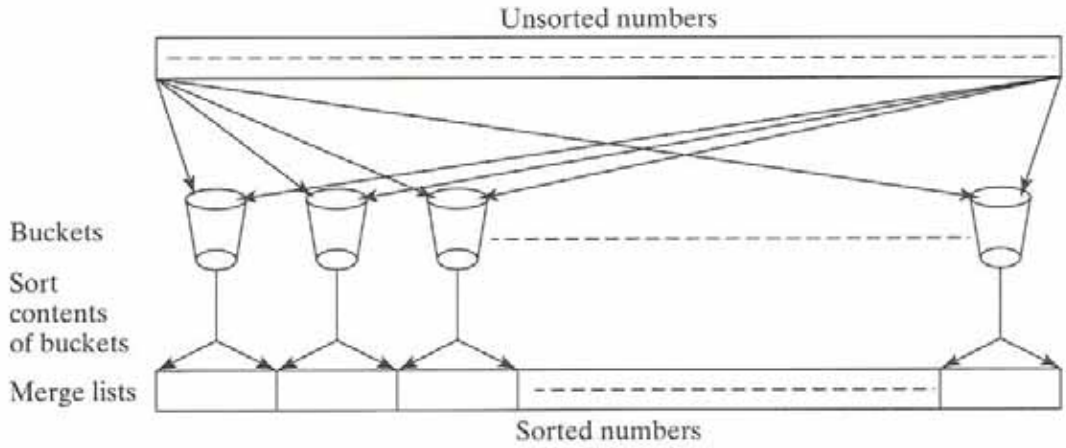


Figure 4.8 Bucket sort.

If $n = km$, where k is a constant, we get a time complexity of $O(n)$. Note that this is much better than the lower bound for sequential compare and exchange sorting algorithms. However, it only applies when the numbers are uniformly distributed.

Parallel Algorithm. Clearly, bucket sort can be parallelized by assigning one processor for each bucket, which reduces the second term in the preceding equation to $(n/p)\log(n/p)$ for p processors (where $p = m$). This implementation is illustrated in Figure 4.9. In this version, each processor examines each of the numbers, so that a great deal of wasted effort takes place. The implementation could be improved by having the processors actually remove numbers from the list into their buckets so that they are not reconsidered by other processors.

We can further parallelize the algorithm by partitioning the sequence into m regions, one region for each processor. Each processor maintains p "small" buckets and separates the numbers in its region into its own small buckets. These small buckets are then "emptied" into the p final buckets for sorting, which requires each processor to send one small bucket to each of the other processors (bucket i to processor i). The overall algorithm is shown in Figure 4.10. Note that this method is a simple partitioning method in which there is minimal work to create the partitions.

The following phases are needed:

1. Partition numbers.
2. Sort into small buckets.
3. Send to large buckets.
4. Sort large buckets.

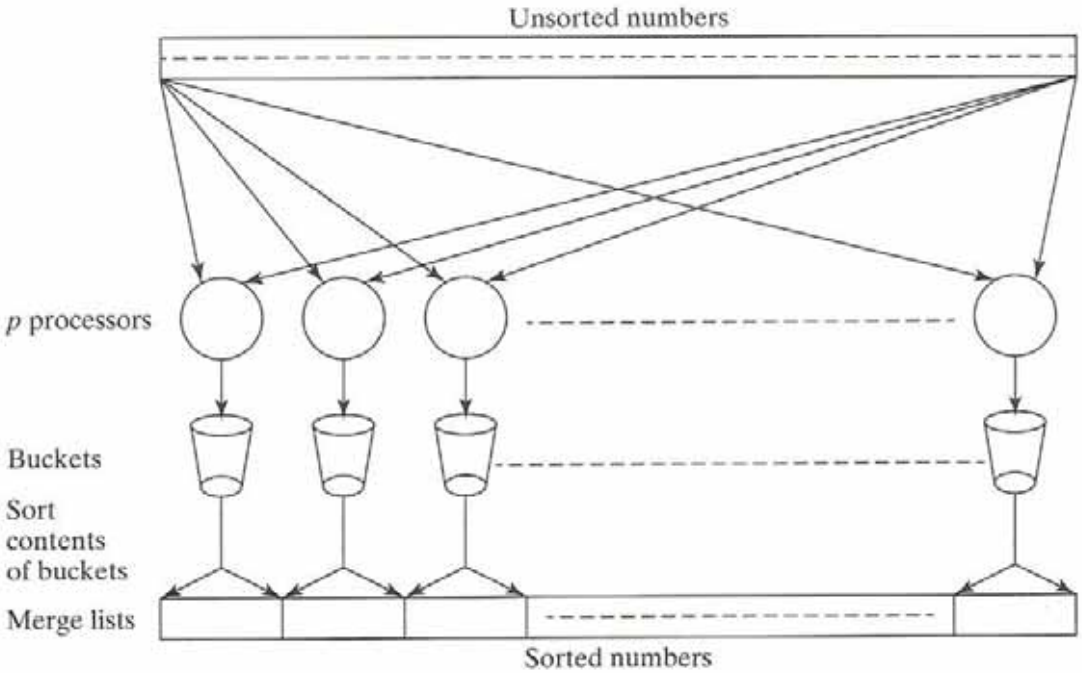


Figure 4.9 One parallel version of bucket sort.

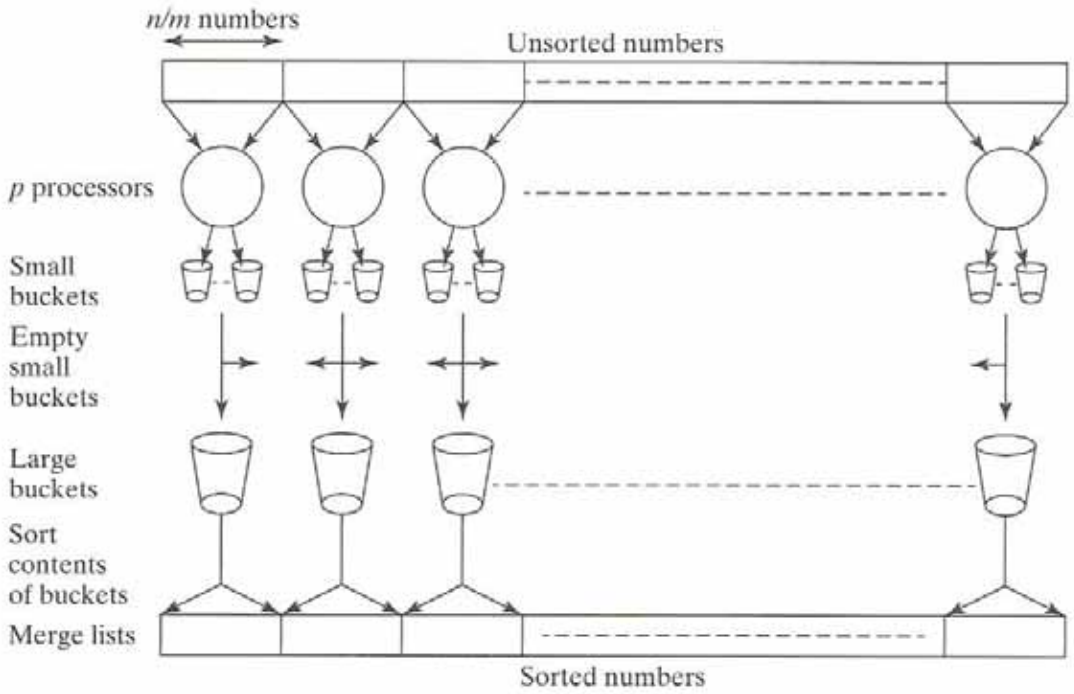


Figure 4.10 Parallel version of bucket sort.

Phase 1 — Computation and Communication. The first step is to send groups of numbers to each processor. Marking a group of numbers into partitions can be done in constant time. This time will be ignored in the overall compassion time. Rather than make the partitions and then send a partition to each processor, a more efficient solution is to simply broadcast all the numbers to each processor and let each processor make its partition. (One must ensure that each partition so created is disjoint but the partitions together include all the numbers.) Using a broadcast or scatter routine, the communication time is:

$$t_{\text{comm1}} = t_{\text{startup}} + nt_{\text{data}}$$

including the communication startup time.

Phase 2 — Computation. To separate each partition of n/p numbers into p small buckets requires the time

$$t_{\text{comp2}} = n/p$$

Phase 3 — Communication. Next, the small buckets are distributed. (There is no computation in Phase 3.) Each small bucket will have about n/p^2 numbers (assuming uniform distribution). Each process must send the contents of $p - 1$ small buckets to other processes (one bucket being held for its own large bucket). Since each process of the p processes must make this communication, we have

$$t_{\text{comm3}} = p(p-1)(t_{\text{startup}} + (n/p^2)t_{\text{data}})$$

if these communications cannot be overlapped in time and individual `send()`s are used. This is the upper bound on this phase of communication. The lower bound would occur if all the communications could overlap, leading to

$$t_{\text{comm3}} = (p-1)(t_{\text{startup}} + (n/p^2)t_{\text{data}})$$

In essence, each processor must communicate with every other processor, and an "all-to-all" mechanism would be appropriate. An all-to-all routine sends data from each process to every other process, as illustrated in Figure 4.11. This type of routine is available in MPI (`MPI_Alltoall()`), which we assume would be implemented more efficiently than using individual `send()`s and `recv()`s. The all-to-all routine will actually transfer the rows of an array to columns, as illustrated in Figure 4.12 (and hence transpose a matrix; see Section 10.2.3).

Phase 4 — Computation. In the final phase, the large buckets are sorted simultaneously. Each large bucket contains about n/p numbers. Thus

$$t_{\text{comp4}} = (n/p)\log(n/p)$$

Overall Execution Time. The overall run time, including communication, is

$$t_p = t_{\text{comm1}} + t_{\text{comp2}} + t_{\text{comm3}} + t_{\text{comp4}}$$

$$t_p = t_{\text{startup}} + nt_{\text{data}} + n/p + (p-1)(t_{\text{startup}} + (n/p^2)t_{\text{data}}) + (n/p)\log(n/p)$$

$$= (n/p)(1 + \log(n/p)) + pt_{\text{startup}} + (n + (p-1)(n/p^2))t_{\text{data}}$$

Speedup Factor. The speedup factor, when compared to sequential bucket sort, is

$$\text{Speedup factor} = \frac{t_s}{t_p} = \frac{n + n\log(n/m)}{(n/p)(1 + \log(n/p)) + pt_{\text{startup}} + (n + (p-1)(n/p^2))t_{\text{data}}}$$

Speedup factor is actually defined where t_s is the time for the best sequential algorithm for the problem. The lower bound for a sequential sorting algorithm using compare and exchange operations and no requirement upon the distribution or special features of the sequence is $n \log n$ steps. However, bucket sort is better than this and is used for t_s, but it has the assumption of uniformly distributed numbers.

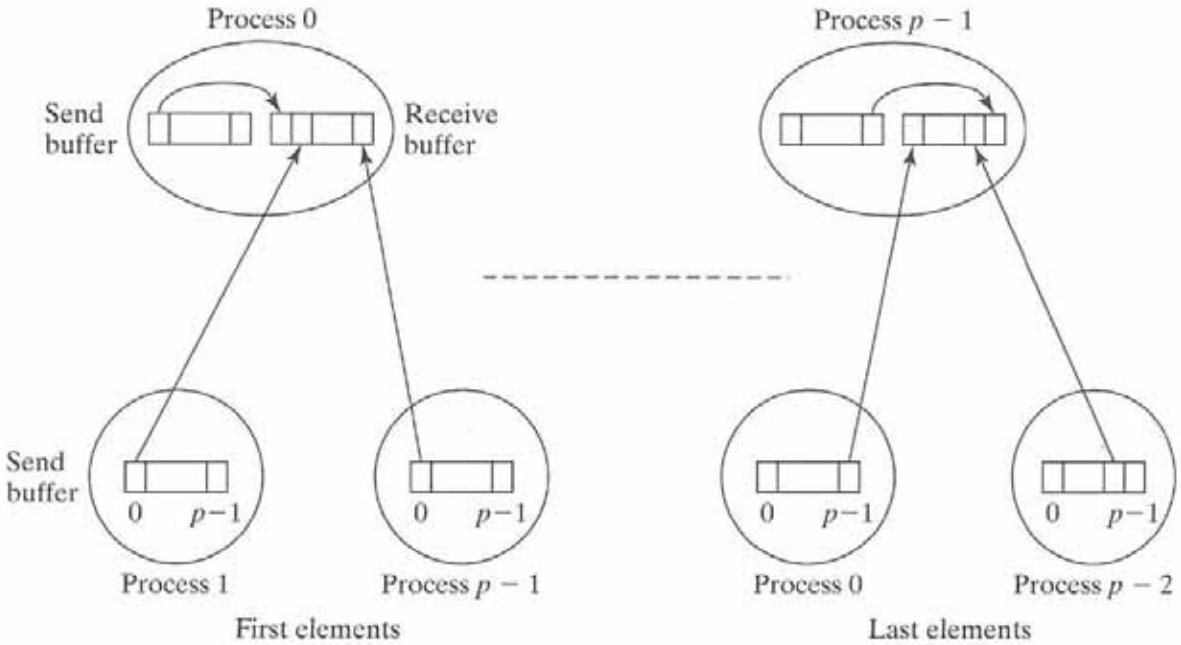


Figure 4.11 All-to-all broadcast.

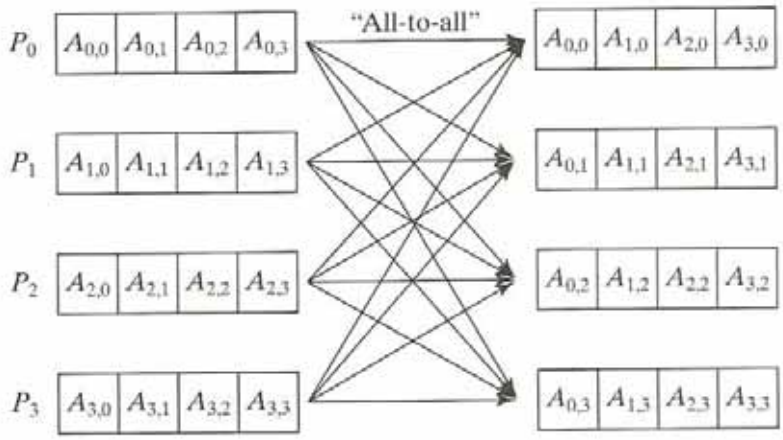


Figure 4.12 Effect of all-to-all on an array.

Computation/communication Ratio. The computation/communication ratio is given by

$$\text{Computation/communication ratio} = \frac{t_{\text{comp}}}{t_{\text{comm}}} = \frac{(n/p)(1+\log(n/p))}{pt_{\text{startup}} + (n+(p-1))(n/p^2)t_{\text{data}}}$$

It is assumed that the numbers are uniformly distributed to obtain these formulas. If the numbers are not uniformly distributed, some buckets would have more numbers than others, and sorting them would dominate the overall computation time. The worst-case scenario would occur when all the numbers fell into one bucket!

4.2.2 Numerical Integration

Previously, we divided a problem and solved each subproblem. The problem was assumed to be divided into equal parts, and partitioning was employed. Sometimes such simple partitioning will not give the optimum solution, especially if the amount of work in each part is difficult to estimate. Bucket sort, for example, is only effective when each region has approximately the same number of numbers. (Bucket sort can be modified to equalize the work.)

A general divide-and-conquer technique divides the region continually into parts and lets an optimization function decide when certain regions are sufficiently divided. Let us take a different example, numerical integration:

$$I = \int_a^b f(x)\,dx$$

To integrate this function (i.e., to compute the "area under the curve"), we can divide the area into separate parts, each of which can be calculated by a separate process. Each region could be calculated using an approximation given by rectangles, as shown in Figure 4.13, where $f(p)$ and $f(q)$ are the heights of the two edges of a rectangular region, and δ is the width (the *interval*). The complete integral can be approximated by the summation of the rectangular regions from a to b. A better approximation can be obtained by aligning the rectangles so that the upper midpoint of each rectangle intersects with the function, as

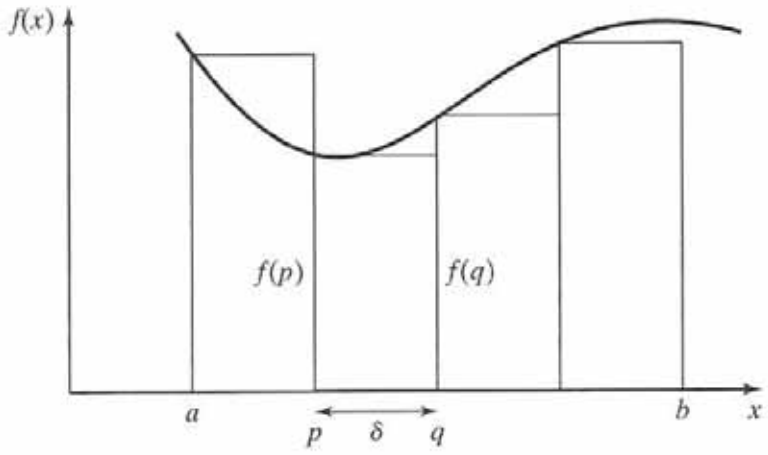


Figure 4.13 Numerical integration using rectangles.

shown in Figure 4.14. This construction has the advantage that the errors on each side of the midpoint end tend to cancel. Another more obvious construction is to use the actual intersections of the vertical lines with the function to create trapezoidal regions, as shown in Figure 4.15. Each region is now calculated as $1/2(f(p) + f(q))\delta$. Such approximate numerical methods for computing a definite integral using a linear combination of values are called *quadrature* methods.

Static Assignment. Let us consider the *trapezoidal* method. Prior to the start of the computation, one process is statically assigned to be responsible for computing each region. By making the interval smaller, we come closer to attaining the exact solution.

Since each calculation is of the same form, the SPMD (single-program multiple-data) model is appropriate. Suppose we were to sum the area from $x = a$ to $x = b$ using p processes numbered 0 to $p - 1$. The size of the region for each process is $(b - a)/p$. To calculate the area in the described manner, a section of SPMD pseudocode could be

Process P_i

```
if (i == master) {                /* read number of intervals required */
   printf("Enter number of intervals ");
   scanf(%d",&n);
}
```

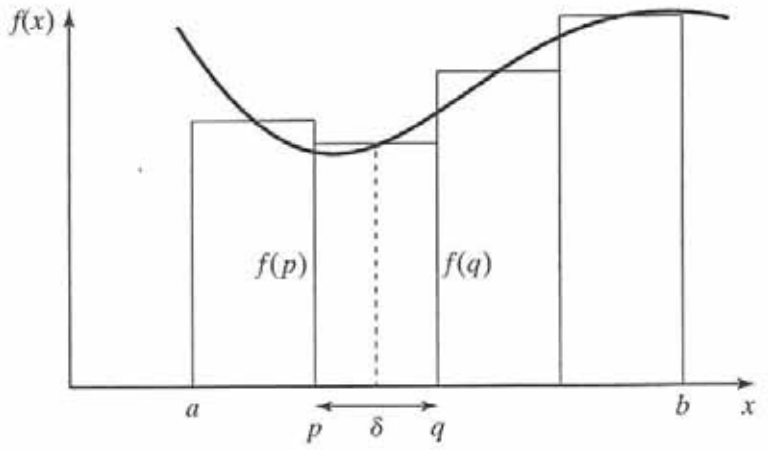


Figure 4.14 More accurate numerical integration using rectangles.

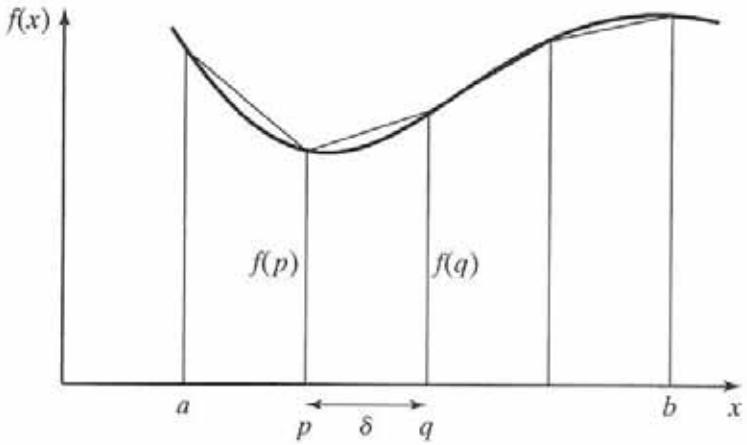


Figure 4.15 Numerical integration using the trapezoidal method.

```
bcast(&n, P_group);              /* broadcast interval to all processes */
region = (b - a)/p;                /* length of region for each process */
start = a + region * i;            /* starting x coordinate for process */
end = start + region;              /* ending x coordinate for process */
d = (b - a)/n;                                      /* size of interval */
area = 0.0;
for (x = start; x < end; x = x + d)
   area = area + 0.5 * (f(x) + f(x+d)) * d;
reduce_add(&integral, &area, P_group);              /* form sum of areas */
```

A reduce operation is used to add the areas computed by the individual processes. For computational efficiency, computing each area is better if written as

```
area = 0.0;
for (x = start; x < end; x = x + d)
   area = area + f(x) + f(x+d);
area = 0.5 * area * d;
```

We assume that the variable `area` does not exceed the allowable maximum value (a possible disadvantage of this variation). For further efficiency, we can simplify the calculation somewhat by algebraic manipulation, as follows:

$$\text{Area} = \frac{\delta(f(a)+f(a+\delta))}{2} + \frac{\delta(f(a+\delta)+f(a+2\delta))}{2} \ldots + \frac{\delta(f(a+(n-1)\delta)+f(b))}{2}$$

$$= \delta\left(\frac{f(a)}{2} + f(a+\delta) + f(a+2\delta) \ldots + f(a+(n-1)\delta) + \frac{f(b)}{2}\right)$$

given n intervals each of width δ. One implementation would be to use this formula for the region handled by each process:

```
area = 0.5 * (f(start) + f(end));
for (x = start + d; x < end; x = x + d)
   area = area + f(x);
area = area * d;
```

Adaptive Quadrature. The methods used so far are fine if we know beforehand the size of the interval δ that will give a sufficiently close solution. We also assumed that a fixed interval is used across the whole region. If a suitable interval is not known, some form of iteration is necessary to converge on the solution. For example, we could start with one interval and reduce it until a sufficiently close approximation is obtained. This implies that the area is recomputed with different intervals, so we cannot simply divide the total region into a fixed number of subregions, as in the summation example.

One approach is for each process to double the number of intervals successively until two successive approximations are sufficiently close. The tree construction could be used for dividing regions. The depth of the tree will be limited by the number of available processes/processors. In our example, it may be possible to allow the tree to grow in an unbalanced fashion as regions are computed to a sufficient accuracy. The phrase *sufficiently close* will depend upon the accuracy of the arithmetic and the application.

Another way to terminate is use three areas, A, B, and C, as shown in Figure 4.16. The computation is terminated when the area computed for the largest of the A and B regions is sufficiently close to the sum of the areas computed for the other two regions. For example, if region B is the largest, terminate when the area of B is sufficiently close to the area of A plus the area of C. Alternatively, we could simply terminate when C is sufficiently small. Such methods are known as *adaptive quadrature* because the solution adapts to the shape of the curve. (Simplified formulas can be derived for adaptive quadrature methods; see Freeman and Phillips, 1992.)

Computations of areas under slowly varying parts of the curve stop earlier than computations of areas under more rapidly varying parts. The spacing, δ, will vary across the interval. The consequence of this is that a fixed process assignment will not lead to the most efficient use of processors. The load-balancing techniques described in Chapter 3, Section 3.2.2, and in more detail in Chapter 7 are more appropriate. We should point out that some care might be needed in choosing when to terminate. For example, the function shown in Figure 4.17 might cause us to terminate early, as two large regions are the same (i.e., $C = 0$).

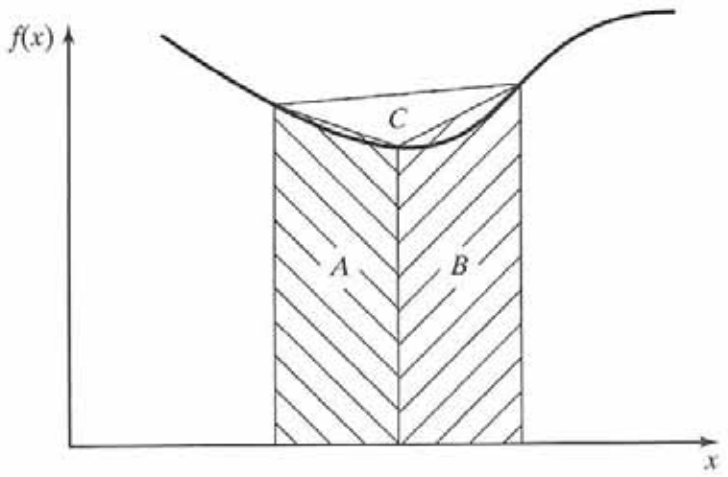


Figure 4.16 Adaptive quadrature construction.

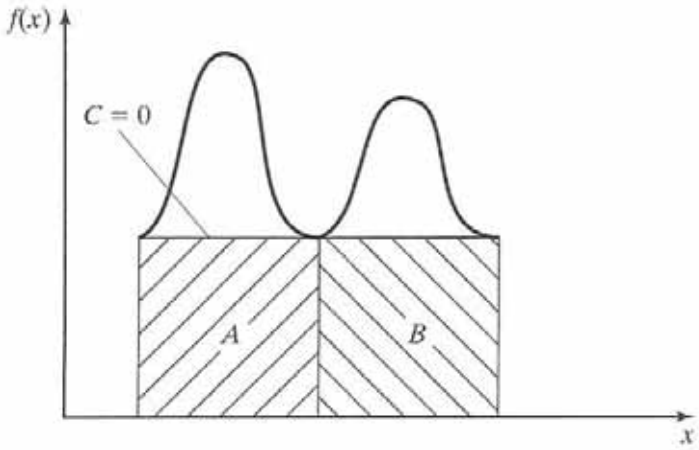


Figure 4.17 Adaptive quadrature with false termination.

4.2.3 *N*-Body Problem

Another problem that can take advantage of divide and conquer is the *N*-body problem. The *N*-body problem is concerned with determining the effects of forces between bodies such as astronomical bodies attracted to each other through gravitational forces. The *N*-body problem also appears in other areas, including molecular dynamics and fluid dynamics. Let us examine the problem in terms of astronomical systems, although the techniques apply to other applications. We provide the basic equations to enable the application to be coded as a programming exercise that could use the same graphic routines as the Mandelbrot problem of Chapter 3 for interesting graphical output.

Gravitational *N*-Body Problem. The objective is to find the positions and movements of bodies in space (e.g., planets) that are subject to gravitational forces from other bodies using Newtonian laws of physics. The gravitational force between two bodies of masses m_a and m_b is given by

$$F = \frac{G m_a m_b}{r^2}$$

where G is the gravitational constant and r is the distance between the bodies. We see that gravitational forces are described by an inverse square law. That is, the force between a pair of bodies is proportional to $1/r^2$, where r is the distance between the bodies. Each body will feel the influence of each of the other bodies according to the inverse square law, and the forces will sum together (taking into account the direction of each force). Subject to forces, a body will accelerate according to Newton's second law:

$$F = ma$$

where m is the mass of the body, F is the force it experiences, and a is the resultant acceleration. All the bodies will move to new positions due to these forces and have new velocities. For a precise numeric description, differential equations would be used (i.e., $F = m dv/dt$ and $v = dx/dt$, where v is the velocity). However, an exact "closed" solution to the *N*-body problem is not known for systems with more than three bodies.

For a computer simulation, we use values at particular times, t_0, t_1, t_2, and so on, the time intervals being as short as possible to achieve the most accurate solution. Let the time

interval be Δt. Then, for a particular body of mass m, the force is given by

$$F = \frac{m(v^{t+1} - v^t)}{\Delta t}$$

and a new velocity

$$v^{t+1} = v^t + \frac{F\Delta t}{m}$$

where v^{t+1} is the velocity of the body at time $t + 1$, and v^t is the velocity of the body at time t. If a body is moving at a velocity v over the time interval Δt, its position changes by

$$x^{t+1} - x^t = v\Delta t$$

where x^t is its position at time t. Once bodies move to new positions, the forces change and the computation has to be repeated.

The velocity is not actually constant over the time interval, Δt, so only an approximate answer is obtained. It can help to use a "leap-frog" computation in which velocity and position are computed alternately:

$$F^t = \frac{m(v^{t+1/2} - v^{t-1/2})}{\Delta t}$$

and

$$x^{t+1} - x^t = v^{t+1/2}\Delta t$$

where the positions are computed for times t, $t + 1$, $t + 2$, and so on, and the velocities are computed for times $t + 1/2$, $t + 3/2$, $t + 5/2$, and so on.

Three-Dimensional Space. Since the bodies are in a three-dimensional space, all values are vectors and have to be resolved into three directions, x, y, and z. In a three-dimensional space having a coordinate system (x, y, z), the distance between the bodies at (x_a, y_a, z_a) and (x_b, y_b, z_b) is given by

$$r = \sqrt{(x_b - x_a)^2 + (y_b - y_a)^2 + (z_b - z_a)^2}$$

The forces are resolved in the three directions, using, for example,

$$F_x = \frac{Gm_a m_b}{r^2}\left(\frac{x_b - x_a}{r}\right)$$

$$F_y = \frac{Gm_a m_b}{r^2}\left(\frac{y_b - y_a}{r}\right)$$

$$F_z = \frac{Gm_a m_b}{r^2}\left(\frac{z_b - z_a}{r}\right)$$

where the particles are of mass m_a and m_b and have the coordinates (x_a, y_a, z_a) and (x_b, y_b, z_b). Finally, the new position and velocity are computed. The velocity can also be

resolved in three directions. For a simple computer solution, we usually assume a three-dimensional space with fixed boundaries. Actually, the universe is continually expanding and does not have fixed boundaries!

Other Applications. Although we describe the problem in terms of astronomical bodies, the concept can be applied to other situations. For example, charged particles are also influenced by each other, in this case according to Coulomb's electrostatic law (also an inverse square law of distance); particles of opposite charge are attracted and those of like charge are repelled. A subtle difference between the problem and astronomical bodies is that charged particles may move away from each other, whereas astronomical bodies are only attracted and thus will tend to cluster.

Sequential Code. The overall gravitational N-body computation can be described by the algorithm

```
for (t = 0; t < tmax; t++) {                /* for each time period */
   for (i = 0; i < N; i++) {                /* for each body */
      F = Force_routine(i);                 /* compute force on ith body */
      v[i]new = v[i] + F * dt / m;          /* compute new velocity and
      x[i]new = x[i] + v[i]new * dt;        /* new position (leap-frog) */
   }
   for (i = 0; i < N; i++) {                /* for each body */
      x[i] = x[i]new;                       /* update velocity and position*/
      v[i] = v[i]new;
   }
}
```

Parallel Code. Parallelizing the sequential algorithm code can use simple partitioning whereby groups of bodies are the responsibility of each processor, and each force is "carried" in distinct messages between processors. However, a large number of messages could result. The algorithm is an $O(N^2)$ algorithm (for one iteration) as each of the N bodies is influenced by each of the other $N-1$ bodies. It is not feasible to use this direct algorithm for most interesting N-body problems where N is very large.

The time complexity can be reduced using the observation that a cluster of distant bodies can be approximated as a single distant body of the total mass of the cluster sited at the center of mass of the cluster, as illustrated in Figure 4.18. This clustering idea can be applied recursively.

Barnes-Hut Algorithm. A clever divide-and-conquer formation to the problem using this clustering idea starts with the whole space in which one cube contains the bodies (or particles). First, this cube is divided into eight subcubes. If a subcube contains no particles, it is deleted from further consideration. If a subcube contains more than one body, it is recursively divided until every subcube contains one body. This process creates an *octtree*; that is, a tree with up to eight edges from each node. The leaves represent cells each containing one body. (We assumed the original space is a cube so that cubes result at each level of recursion, but other assumptions are possible.)

For a two-dimensional problem, each recursive subdivision will create four subareas and a *quadtree* (a tree with up to four edges from each edge; see Section 4.1.3). In general,

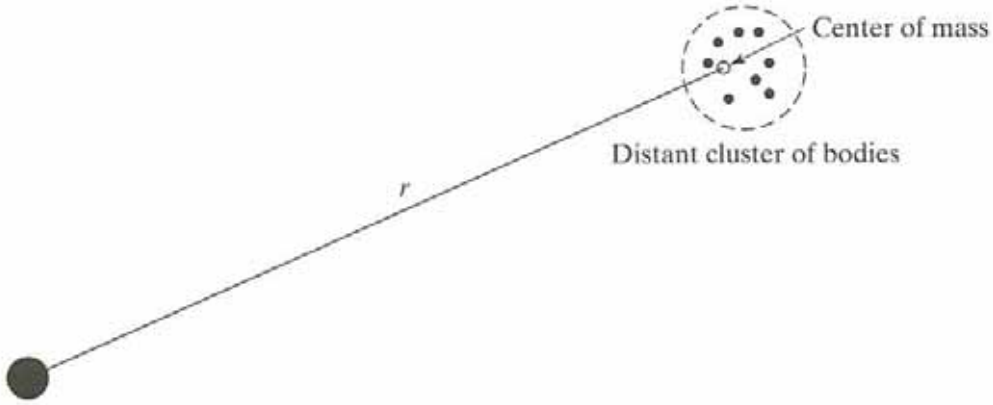


Figure 4.18 Clustering distant bodies.

the tree will be very unbalanced. Figure 4.19 illustrates the decomposition for a two-dimensional space (which is easier to draw) and the resultant quadtree. The three-dimensional case follows the same construction except that it has up to eight edges from each node.

In the *Barnes-Hut algorithm* (Barnes and Hut, 1986), after the tree has been constructed, the total mass and center of mass of the subcube is stored at each node. The force on each body can then be obtained by traversing the tree, starting at the root, stopping at a node when the clustering approximation can be used for the particular body, and otherwise continuing to traverse the tree downward. In astronomical N-body simulations, a simple criterion for when the approximation can be made is as follows. Suppose the cluster is enclosed in a cubic volume given by the dimensions $d \times d \times d$, and the distance to the center of mass is r. Use the clustering approximation when

$$r \geq \frac{d}{\theta}$$

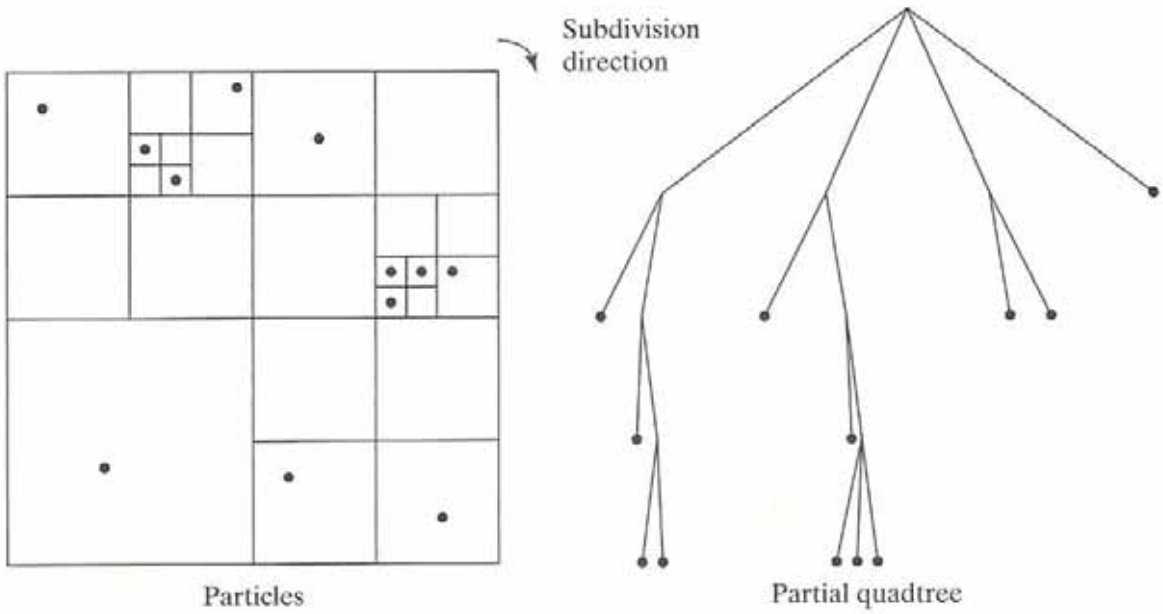


Figure 4.19 Recursive division of two-dimensional space.

where θ is a constant typically 1.0 or less (θ is called the opening angle). This approach can substantially reduce the computational effort.

Once all the bodies have been given new positions and velocities, the process is repeated for each time period. This means that the whole octtree must be reconstructed for each time period (because the bodies have moved). Constructing the tree requires a time of $O(n \log n)$, and so does computing all the forces, so that the overall time complexity of the method is $O(n \log n)$ (Barnes and Hut, 1986).

The algorithm can be described by the following:

```
for (t = 0; t < tmax; t++) {        /* for each time period */
   Build_Octtree();                 /* construct Octtree (or Quadtree) */
   Tot_Mass_Center();               /* compute total mass & center /*
   Comp_Force();                    /* traverse tree/computing forces */
   Update();                        /* update position/velocity */
}
```

The `Build_Octtree()` routine can be constructed from the positions of the bodies, considering each body in turn. The `Tot_Mass_Center()` routine must traverse the tree, computing the total mass and center of mass at each node. This could be done recursively. The total mass, M, is given by the simple sum of the total masses of the children:

$$M = \sum_{i=0}^{7} m_i$$

where m_i is the total mass of the ith child. The center of mass, C, is given by

$$C = \frac{1}{M} \sum_{i=0}^{7} (m_i \times c_i)$$

where the positions of the centers of mass have three components, in the x, y, and z directions. The `Comp_Force()` routine must visit nodes ascertaining whether the clustering approximation can be applied to compute the force of all the bodies in that cell. If the clustering approximation cannot be applied, the children of the node must be visited.

The octtree will, in general, be very unbalanced, and its shape changes during the simulation. Hence, a simple static partitioning strategy will not be very effective in load balancing. A better way of dividing the bodies into groups is called *orthogonal recursive bisection* (Salmon, 1990). Let us describe this method in terms of a two-dimensional square area. First, a vertical line is found that divides the area into two areas, each with an equal number of bodies. For each area, a horizontal line is found that divides it into two areas, each with an equal number of bodies. This is repeated until there are as many areas as processors, and then one processor is assigned to each area. An example of the division is illustrated in Figure 4.20.

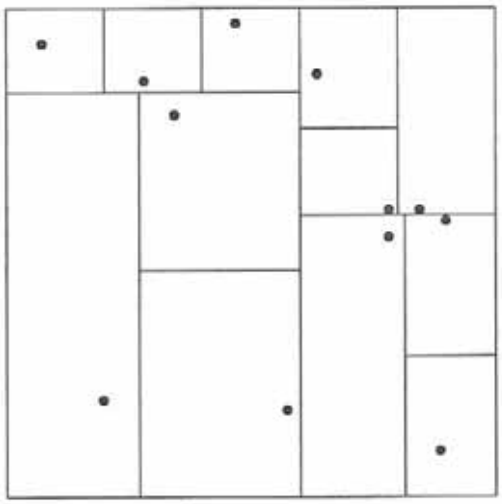


Figure 4.20 Orthogonal recursive bisection method.

4.3 SUMMARY

This chapter introduced the following concepts:

- Partitioning and divide-and-conquer as the basis for parallel computing techniques
- Tree constructions
- Examples of partitioning and divide-and-conquer problems — namely, bucket sort, numerical integration, and the N-body problem

FURTHER READING

The divide-and-conquer technique is described in many data structure and algorithms texts (e.g., Cormen Leiserson and Rivest, 1990). As we have seen, this technique results in a tree structure. It is possible to construct a multiprocessor with a tree network that would then be amenable to divide-and-conquer problems. One or two tree network machines have been constructed with the thought that most applications can be formulated as divide and conquer. However, as we have seen in Chapter 1, trees can be embedded into meshes and hypercubes so that it is not necessary to have tree network. Mapping divide-and-conquer algorithms onto different architectures is the subject of research papers like the one by Lo and Rajopadhye (1990).

Once a problem is partitioned, a scheduling algorithm is appropriate in some contexts for allocating processors to partitions or processes. One text dedicated to partitioning and scheduling is Sarkar (1989). Mapping (static scheduling) is not considered in this text. However, dynamic load balancing, in which tasks are assigned to processors during the execution of the program, is considered in Chapter 7.

Bucket sort is described in texts on sorting algorithms (see Chapter 9) and can be found specifically in Lindstrom (1985) and Wagner and Han (1986). Numerical evaluation of integrals in the context of parallel programs can be found in Freeman and Phillips (1992),

Gropp, Lusk, and Skjellum (1999), and Smith (1993) and is often used as a simple application of parallel programs. The original source for the Barnes-Hut algorithm is Barnes and Hut (1986). Other papers include Bhatt et al. (1992) and Warren and Salmon (1992). Liu and Wu (1997) consider programming the algorithm in C++. Apart from the Barnes-Hutt divide-and-conquer algorithm, another approach is the fast multipole method (Greengard and Rokhlin, 1987). Hybrid methods exist.

BIBLIOGRAPHY

AHO, A. V., J. E. HOPCROFT, AND J. D. ULLMAN (1974), *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA.

BARNES, J. E., AND P. HUT (1986), "A Hierarchical O(Nlog$N$) Force Calculation Algorithm," *Nature*, Vol. 324, No. 4 (December), pp. 446–449.

BHATT, S., M. CHEN, C. Y. LIN, AND P. LIU (1992), "Abstractions for Parallel N-Body Simulations," *Proc. Scalable High Performance Computing Conference*, pp. 26–29.

BLELLOCH, G. E. (1996), "Programming Parallel Algorithms," *Comm. ACM*, Vol. 39, No. 3, pp. 85–97.

BOKHARI, S. H. (1981), "On the Mapping Problem," *IEEE Trans. Comput.*, Vol. C-30, No. 3, pp. 207–214.

CORMEN, T. H., C. E. LEISERSON, AND R. L. RIVEST (1990), *Introduction to Algorithms*, MIT Press, Cambridge, MA.

FREEMAN, T. L., AND C. PHILLIPS (1992), *Parallel Numerical Algorithms*, Prentice Hall, London.

GREENGARD, L., AND V. ROKHLIN (1987), "A Fast Algorithm for Particle Simulations," *J. Comp. Phys.*, Vol. 73, pp. 325–348.

GROPP, W., E. LUSK, AND A. SKJELLUM (1999), *Using MPI Portable Parallel Programming with the Message-Passing Interface*, MIT Press, Cambridge, MA.

JÁJÁ, J. (1992), *An Introduction to Parallel Algorithms*, Addison Wesley, Reading, MA.

LINDSTROM, E. E. (1985), "The Design and Analysis of BucketSort for Bubble Memory Secondary Storage," *IEEE Trans. Comput.*, Vol. C-34, No. 3, pp. 218–233.

LIU, P., AND J.-J. WU (1997), "A Framework for Parallel Tree-Based Scientific Simulations," *Proc. 1997 Int. Conf. Par. Proc.*, pp. 137–144.

LO, V. M., AND S. RAJOPADHYE (1990), "Mapping Divide-and-Conquer Algorithms to Parallel Architectures," *Proc. 1990 Int. Conf. Par. Proc.*, Part III, pp. 128–135.

MILLER, R., AND Q. F. STOUT (1996), *Parallel Algorithms for Regular Architectures: Meshes and Pyramids*, MIT Press, Cambridge, MA.

PREPARATA, F. P., AND M. I. SHAMOS (1985), *Computational Geometry: An Introduction*, Springer-Verlag, NY.

SALMON, J. K. (1990), *Parallel Hierarchical N-Body Methods*, Ph.D. thesis, California Institute of Technology.

SARKAR, V. (1989), *Partitioning and Scheduling Parallel Programs for Multiprocessing*, MIT Press, Cambridge, MA.

SMITH, J. R. (1993), *The Design and Analysis of Parallel Algorithms*, Oxford University Press, Oxford, England.

WAGNER, R. A., AND Y. HAN (1986), "Parallel Algorithms for Bucket Sorting and Data Dependent Prefix Problem," *Proc. 1986 Int. Conf. Par. Proc.*, pp. 924–929.

WARREN, M., AND J. SALMON (1992), "Astrophysical *N*-Body Simulations Using Hierarchical Tree Data Structures," *Proc. Supercomputing 92*, IEEE CS Press, Los Alamitos, pp. 570–576.

PROBLEMS

Scientific/Numerical

4-1. Write a program that will prove that the maximum speedup of adding a series of numbers using a simple partition described in Section 4.1.1 is $p/2$, where there are p processes.

4-2. Using the equations developed in Section 4.1.1 for partitioning a list of numbers into m partitions that are added separately, show that the optimum value for m to give the minimum parallel execution time is when $m = \sqrt{p/(1 + t_{\text{startup}})}$, where there are p processors. (Clue: Differentiate the parallel execution time equation.)

4-3. Section 4.1.1 gives three implementations of adding numbers, using separate `send()`s and `recv()`s, using a broadcast routine with separate `recv()`s to return partial results, and using scatter and reduce routines. Write parallel programs for all three implementations, instrumenting the programs to extract timing information (Chapter 2, Section 2.3.4), and compare the results.

4-4. Suppose the structure of a computation consists of a binary tree with n leaves (final tasks) and $\log n$ levels. Each node in the tree consists of one computational step. What is the lower bound of the execution time if the number of processors is less than n?

4-5. Analyze the divide-and-conquer method of assigning one processor to each node in a tree for adding numbers (Section 4.1.2) in terms of communication, computation, overall parallel execution time, speedup, and efficiency.

4-6. Complete the parallel pseudocode given in Section 4.1.2 for the (binary) divide-and-conquer method for all eight processes.

4-7. Develop the equations for computation and communication times for m-ary divide and conquer, following the approach used in Section 4.1.2.

4-8. Develop a divide-and-conquer algorithm that finds the smallest value in a set of n values in $O(\log n)$ steps using $n/2$ processors. What is the time complexity if there are fewer than $n/2$ processors?

4-9. Write a parallel program with a time complexity of $O(\log n)$ to compute the polynomial

$$f = a_0x^0 + a_1x^1 + a_2x^2 + \ldots + a_{n-1}x^{n-1}$$

to any degree, n, where the a's, x, and n are input.

4-10. Write a parallel program that uses a divide-and-conquer approach to find the first zero in a list of integers stored in an array. Use 16 processes and 256 numbers.

4-11. Write parallel programs to compute the summation of n integers in each of the following ways and assess their performance. Assume that n is a power of 2.

(a) Partition the n integers into $n/2$ pairs. Use $n/2$ processes to add together each pair of integers resulting in $n/2$ integers. Repeat the method on the $n/2$ integers to obtain $n/4$ integers and continue until the final result is obtained. (This is a binary tree algorithm.)

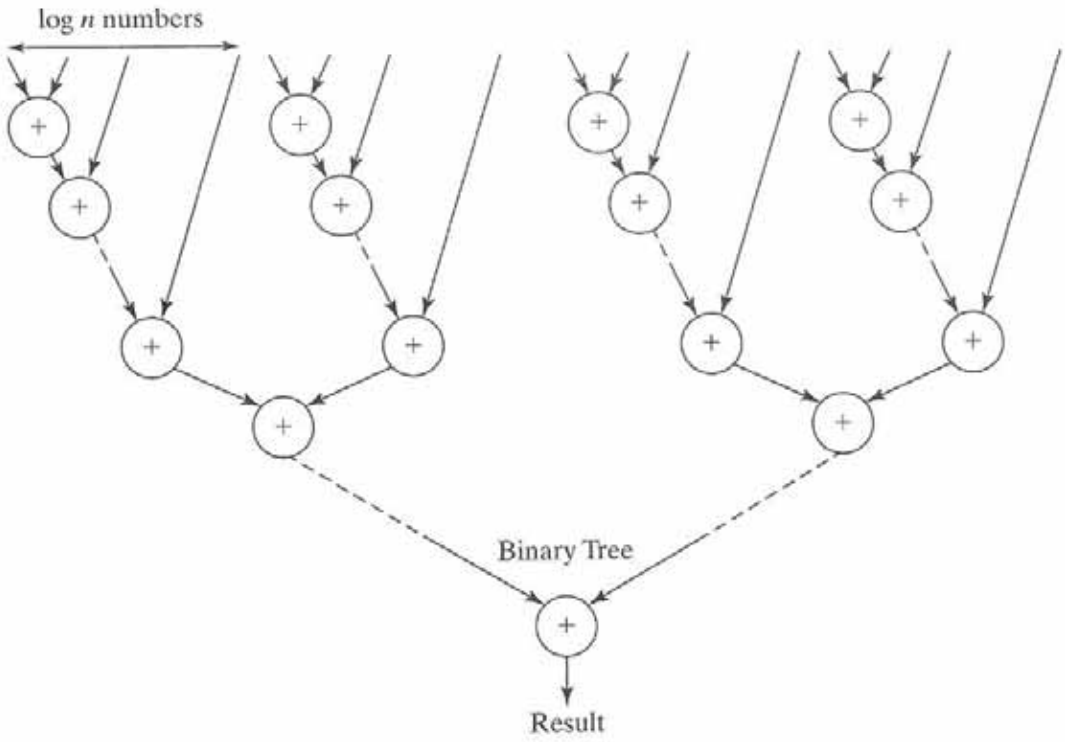


Figure 4.21 Process diagram for Problem 4-12(b).

(b) Divide the n integers into $n/\log n$ groups of $\log n$ numbers each. Use $n/\log n$ processes, each adding the numbers in one group sequentially. Then add the $n/\log n$ results using method (a). This algorithm is shown in Figure 4.21.

4-12. Write parallel programs to compute $n!$ in each of the following ways and assess their performance. The number, n, may be odd or even but is a positive constant.

(a) Compute $n!$ using two concurrent processes, each computing approximately half of the complete sequence. A master process then combines the two partial results.

(b) Compute $n!$ using a producer process and a consumer process connected together. The producer produces the numbers 1, 2, 3, ... n in sequence. The consumer accepts the sequence of numbers from the producer and accumulates the result; i.e., $1 \times 2 \times 3 \ldots$.

4-13. Write a divide-and-conquer parallel program that determines whether the number of 1's in a binary file is even or odd (i.e., create a parity checker). Modify the program so that a bit is attached to the contents of the file, and set to a 0 or a 1 to make the number of 1's even (a parity generator).

4-14. Bucket sort and its parallel implementations suffer for poor performance if the numbers are not uniformly distributed, because more numbers will fall into the same bucket for subsequent sorting. Modify the algorithm so that the regions that each bucket collects are altered. This could be done as the algorithm is executed or before in a preprocessing step. Implement your algorithm.

4-15. One way to compute π is to compute the area under the curve $f(x) = 4/(1 + x^2)$ between 0 and 1, which is numerically equal to π. Write a parallel program to calculate π this way using 10 processes. Another way to compute π is to compute the area of a circle of radius $r = 1$ (i.e., $\pi r^2 = \pi$). Determine the appropriate equation for a circle, and write a parallel program to compute π this way. Comment on the two ways of computing π.

4-16. Derive a formula to evaluate numerically an integral using the adaptive quadrature method described in Section 4.2.2. Use the approach given for the trapezoidal method.

4-17. Using any method, write a parallel program that will compute the integral

$$I = \int_{0.01}^{1} \left(x + \sin\left(\frac{1}{x}\right) \right) dx$$

4-18. Write a static assignment parallel program to compute π using the formula

$$\int_0^1 \sqrt{1 - x^2}\, dx = \frac{\pi}{4}$$

using each of the following ways:

1. Rectangular decomposition, as illustrated in Figure 4.13
2. Rectangular decomposition, as illustrated in Figure 4.14
3. Trapezoidal decomposition, as illustrated in Figure 4.15

Evaluate each method in terms of speed and accuracy.

4-19. Find the zero crossing of a function by a bisection method. In this method, two points on the function are computed, say $f(a)$ and $f(b)$, where $f(a)$ is positive and $f(b)$ is negative. The function must cross somewhere between a and b, as illustrated in Figure 4.22. By

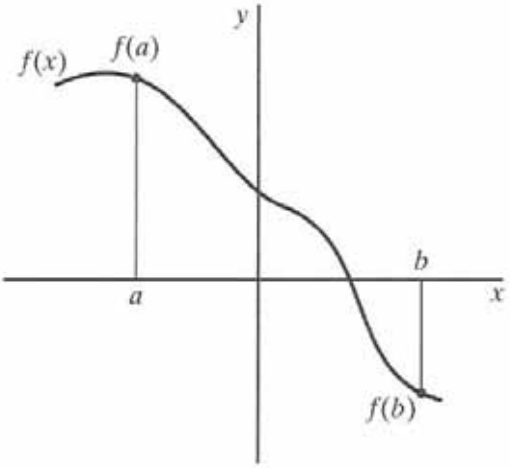


Figure 4.22 Bisection method for finding the zero crossing location of a function.

successively dividing the interval, the exact location of the zero crossing can be found. Write a divide-and-conquer program that will find the zero crossing locations of the function $f(x) = x^2 - 3x + 2$. (This function has two zero crossing locations, $x = 1$ and $x = 2$.)

4-20. Write a parallel program to integrate a function using Simpson's rule, which is given as follows:

$$I = \int_a^b f(x)\, dx =$$

$$\frac{\delta}{3}[f(a) + 4f(a+\delta) + 2f(a+2\delta) + 4f(a+3\delta) + 2f(a+4\delta) + \ldots 4f(a+(n-1)\delta) + f(b)]$$

where δ is fixed [$\delta = (b - a)/n$ and n must be even]. Choose a suitable function (or arrange it so that the function can be input).

4-21. Write a sequential program and a parallel program to simulate an astronomical N-body system, but in two dimensions. The bodies are initially at rest. Their initial positions and masses are to

be selected randomly (using a random-number generator). Display the movement of the bodies using the graphical routines used for the Mandelbrot program found in http://www.cs.uncc.edu/par_prog, or otherwise, showing each body in a color and size to indicate its mass.

4-22. Develop the N-body equations for a system of charged particles (e.g., free electrons and positrons) that are governed by Coulumb's law. Write a sequential and a parallel program to model this system, assuming that the particles lie in a two-dimensional space. Produce graphical output showing the movement of the particles. Provide your own initial distribution and movement of particles and solution space.

4-23. (Research problem) Given a set of n points in a plane, develop an algorithm and parallel program to find the points that are on the perimeter of the smallest region containing all of the points, and join the points, as illustrated in Figure 4.23. This problem is known as the planar convex hull problem and can be solved by a recursive divide-and-conquer approach very similar to quicksort, by recursively splitting regions into two parts using "pivot" points. There are several sources for information on the planar convex hull problem, including Blelloch (1996), Preparata and Shamos (1985), and Miller and Stout (1996).

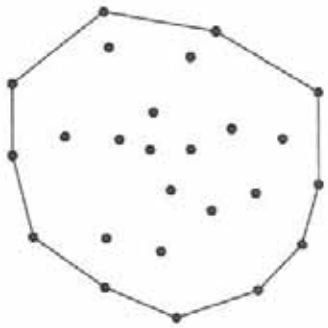


Figure 4.23 Convex hull (Problem 4-23).

Real Life

4-24. Write a sequential and a parallel program to model the planets around the sun (or another astronomical system). Produce graphical output showing the movement of the planets. Provide your own initial distribution and movement of planets. (Use real data if available.)

4-25. A major bank in your state processes an average of 30 million checks a day for its 2 million customer accounts. One time-consuming problem is that of sorting the checks into individual customer-account bundles so they can be returned with the monthly statements. (Yes, the bank handles check sorting for several client banks in addition to its own.) The bank has been using a very fast mainframe-based check sorter and the quicksort method. However, you have told the bank that you know of a way to use N smaller computers in parallel; each will sort $1/N$th of the 30 million checks, and then the partial sorts will be merged into a single sorted result. Before investing in the new technology, the bank hires you as a consultant to simulate the process using message-passing parallel programming. Under the following assumptions, simulate this new approach for the bank.

Assumptions:

1. Each check has three identification numbers: a nine-digit bank-identification number, a nine-digit account-identification number, and a three-digit check number (leading zeros are not printed or shown).

2. All checks with the same bank-identification number are to be sorted by customer account for transmittal to the client bank.

Estimate the speedup if N is 10 and if N is 1000. Estimate the percentage of time spent in communications versus time spent in processing.

4-26. Sue, 21 years old, comes from a very financially astute family. She has been watching her parents save and invest for several years now, reads the *Wall Street Journal* daily in the university library (for free!), and has concluded that she will not be able to rely on social security when she retires in 49 years. For graduation from college, her parents got her a CD-ROM containing historical daily closing prices covering every exchange-listed security, from January 1, 1900 to the end of last month.

For simplicity you may think of the data on the CD-ROM as organized into date/symbol/closing price records for each of the 358,000 securities that have been listed since 1900. (Only a fraction are listed at any given date; firms go out of business and new ones start daily.) Similarly, you may assume that the format of a record is given by

date	Last three digits of the year, followed by the "Julian date" (where January 15 is Julian 15, February 1 is Julian 32, etc.)
symbol	Up to 10 characters, such as PCAI, KAUFX, or IBM.AZ, representing a NASDAQ stock (PCA International), a mutual fund (Kaufman Aggressive Growth), and an option to buy IBM stock at a certain price for a certain length of time, respectively.
closing price	Three integers, X (representing a whole number of dollars per unit), Y (representing the numerator of a fractional dollar per unit), and Z (representing the denominator of a fractional dollar per unit).

For example, "996033/PCAI/10/3/4" indicates that on February 2, 1996, PCA International stock closed at $10.75 per share. Sue wants to know how many of the stocks that were listed as of last month's end have had 50 or more consecutive trading days in which they closed either unchanged from the previous day or at a higher price, anytime in the CD-ROM's recorded history.

4-27. The more Samantha recalled her grandfather's stories about the time he won the 1963 World Championship Dominos Match, the more she wanted to improve her skills at the game. She had a basic problem, though; she had no playing partners left, having already improved to the point where she consistently won every game against the few friends who still remained!

Samantha knew that computerized versions of go, chess, bridge, poker, and checkers had been developed, and saw no reason someone skilled in the science of computers could not do the same for dominos. One of her computer science professors at the second campus of the University of Canada, U-Can-2, had told her she could do anything she wanted (within theoretical limits, of course), and she really wanted to win that next world championship!

Pulling out her slow, nearly ancient 2-cubed Itanium (2 GHz, 2GB RAM, 2 TB hard disk), she quickly developed a straightforward single-processor simulator that she could practice against. The basic outline of her approach was to have the program compare every one of its pieces to the pieces already played in order to determine the computer's best move. This appeared to involve so many computations, including rotations and trial placements of pieces that Samantha found herself waiting for the program to produce the computer's next move, and becoming as bored with its game performance as with that of her old friends. Thus, she is seeking your assistance in developing a parallel-processor version.

1. Outline her single-processor algorithm.
2. Outline your parallel-processor algorithm.

3. Estimate the speedup that could be obtained if you were to network 50 old computers like hers. Then make a recommendation to her about either going ahead with the task or spending $800 to buy the latest processor, reputed to be at least 50 times faster than her old Itanium for these types of simulations: the new 14GHz dual processor Octium with its standard 1024-bit front-side data bus and 2-way simultaneous access to its 16TB of 0.5ns main memory.

4-28. Area, Inc., provides a numerical integration service for several small engineering firms in the region. When any of these firms has a continuous function defined over a domain and is unable to integrate it, Area, Inc., gets the call. You have just been hired to help Area, Inc., improve its slow delivery of computed integration results. Area, Inc. has lost money each year of its existence and is so "nonprofit" that payment of next week's payroll is in question. Given your desire to continue eating (and for that Area, Inc., has to pay you), you have considerable incentive to help Area, Inc.

Given also that you have a considerable background in parallel computing, you recognize the problem immediately: Area, Inc., has been using a single processor to implement a standard numerical integration algorithm.

Step 1: Divide the independent axis into N even intervals.

Step 2: Approximate the area under the function in any interval (its integral over the interval), by the product of the interval width times the function value when it is evaluated at the left edge of the interval.

Step 3: Add up all N approximations to get the total area.

Step 4: Divide the interval width in half.

Step 5: Repeat steps $1-4$ until the total from the ith repetition differs from the $(i-1)th$ repetition by less than 0.001 percent of the magnitude of the ith total.

Since your manager is skeptical about newfangled parallel computing approaches, she wants you to simulate two different machine configurations: two processors in the first, and eight processors in the second. She has told you that a successful demonstration is key to being able to buy more processors and to your getting paid next week.

4-29. The Search for Extra-Terrestrial Intelligence (SETI) project employs millions of computers to analyze radio-telescope signals from the Arecibo Observatory in Puerto Rico. Each is given a time- and frequency-limited portion of the signals recorded by the world's largest radio telescope and asked to perform a computationally intense analysis to determine the likelihood of that portion containing intelligent communications from another life form. This is clearly an example of a divide-and-conquer approach.

Discuss how this approach could be applied to the more local problem of analysis of radio-frequency communications in the world's anti-terrorism struggle.

4-30. Rafic loves to solve crossword puzzles. Lately, he has begun to create them when he cannot find a suitably challenging one on which to work. This process has two parts: laying out the pattern of open squares into which individual letters will go, and blackened squares that form breaks between words or phrases. He has a word and phrase dictionary containing over 100,000 words and phrases ranging from ancient Greek and Roman references to such modern things as the Itanium-III and pico-technology. Rafic needs your help.

Develop a sequential algorithm that will produce a crossword puzzle of size $N \times N$, and then convert the algorithm to its parallel counterpart.

4-31. On occasion, beginning computer science students are tempted to copy work done by others and submit it as their own, a practice, known as plagiarism, that typically results in severe

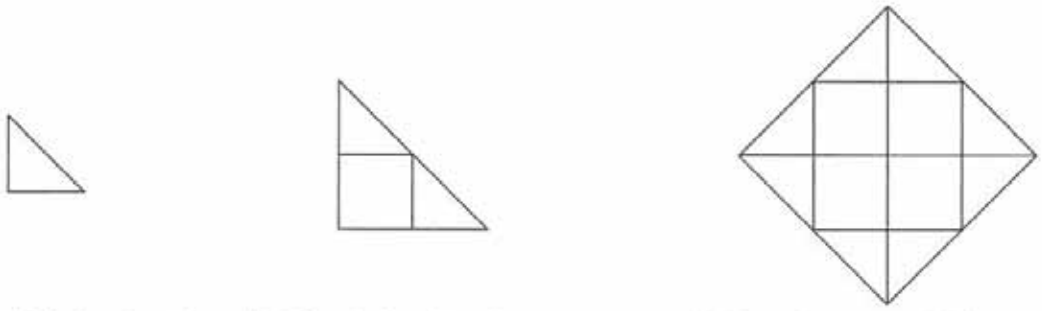
academic penalties when detected. Sometimes the more creative students will make small changes to the work before submitting it: changing variable names, changing indentation, and sometimes even changing looping structures (substituting a "while" for a "for"). For a typical first-year course with 400 students, this requires approximately 80,000 program comparisons per programming assignment. As a result, total checking is rarely done.

(Easy) Develop a parallel approach using N computers that can exhaustively check for exact duplicates among the 400 submissions on a typical programming assignment.

(Harder) Pre-process each program to tokenize the variable names, thereby converting each program to a standard set of variable names. Then apply the approach from the "easy" part.

(Hardest) Pre-process each program to put all loops into the same structure: a "for". Then implement both of the earlier parts.

4-32. Sarah, a friend of Tom's has puzzle-creation as her hobby. On her home CAD system tied to the computer-controlled milling machine in her workshop she designed three elementary shapes and fabricated thousands of each. The first is a right triangle whose shortest sides are 1 unit in length. The second is a square whose edges are 1 unit in length. The third is a combination of a rectangle 4 units by 13 units attached to a right triangle whose shortest sides are 4 units. Given an arbitrary combination of pieces (F of type first, S of type second, and T of type third), Sarah needs you to develop and implement an algorithm that will determine the area of the largest right triangle that can be formed by placing some or all of these $F + S + T$ pieces together ... initially as a sequential algorithm and then as a parallel algorithm with N computers working on the solution. Sample combinations of these pieces are shown in Figure 4.24 to get you started.



(a) Single triangle. (b) Pair of triangles plus one square. (c) Four instances of (b), each area is 0.5 sq units area is 2 sq units rotated 90 degrees to form a square whose sides are each $2\sqrt{2}$ in length, area is 8 sq units.

Figure 4.24 Triangles (Problem 4-32).

Chapter 5

Pipelined Computations

In this chapter, we present a parallel processing technique called *pipelining*, which is applicable to a wide range of problems that are partially sequential in nature; that is, a sequence of steps must be undertaken. Hence, pipelining can be used to parallelize sequential code. Certain requirements are necessary for improved performance, as will be outlined.

5.1 PIPELINE TECHNIQUE

In the pipeline technique, the problem is divided into a series of tasks that have to be completed one after the other. In fact, this is the basis of sequential programming. In pipelining, each task is executed by a separate process or processor, as shown in Figure 5.1. We sometimes refer to each pipeline process as a pipeline *stage*. Each stage contributes to the overall problem and passes on information that is needed for subsequent stages. This parallelism can be viewed as a form of *functional decomposition*. The problem is divided into separate functions that must be performed, but in this case, the functions are performed in succession. As we shall see, the input data is often broken up and processed separately.

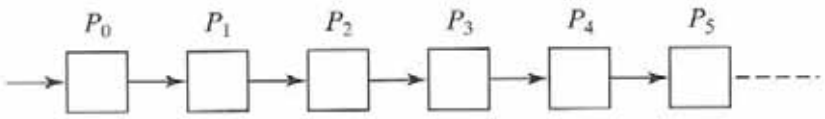


Figure 5.1 Pipelined processes.

As an example of a sequential program that can be formulated as a pipeline, consider a simple loop:

```
for (i = 0; i < n; i++)
   sum = sum + a[i];
```

which adds all the elements of array `a` to an accumulating sum. The loop could be "unrolled" to yield

```
sum = sum + a[0];
sum = sum + a[1];
sum = sum + a[2];
sum = sum + a[3];
sum = sum + a[4];
      ⋮
```

One pipeline solution would have a separate stage for each statement, as shown in Figure 5.2. Each stage accepts the accumulating sum on its input, s_{in}, and one element of `a[]` on its input, a, and produces the new accumulating sum on its output, s_{out}. Therefore, stage i performs

```
sout = sin + a[i];
```

Instead of simple statements, a series of functions can be performed in a pipeline fashion. A frequency filter is a more realistic example in which the problem is divided into a series of functions (functional decomposition). The objective here is to remove specific frequencies (say the frequencies f_0, f_1, f_2, f_3, etc.) from a (digitized) signal, $f(t)$. The signal could enter the pipeline from the left, as shown in Figure 5.3. Each stage is responsible for removing one of the frequencies.

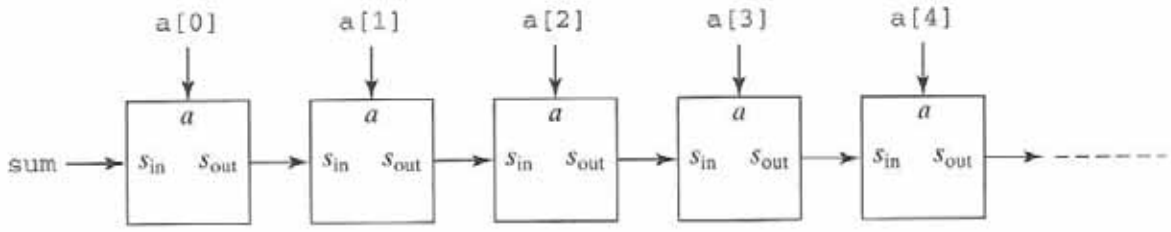


Figure 5.2 Pipeline for an unrolled loop.

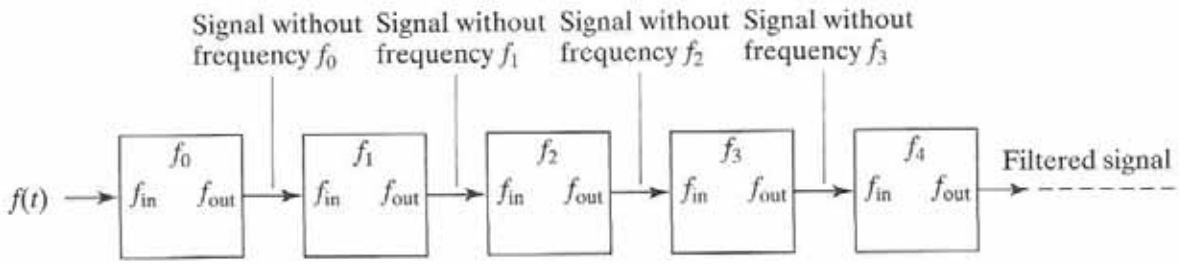


Figure 5.3 Pipeline for a frequency filter.

A similar application is to recognize certain frequencies in a signal. In home and professional sound systems, for example, there often is a display showing the specific frequencies in the audio output. Each stage of a pipeline could recognize one frequency and display its amplitude as part of a frequency-amplitude histogram. Problem 5-13 explores this application.

Given that the problem can be divided into a series of sequential tasks, the pipelined approach can provide increased speed under the following three types of computations:

1. If more than one instance of the complete problem is to be executed
2. If a series of data items must be processed, each requiring multiple operations
3. If information to start the next process can be passed forward before the process has completed all its internal operations.

We will identify these three solutions as Type 1, Type 2, and Type 3.

The Type 1 arrangement is utilized widely in the internal hardware design of computers. It also appears in simulation exercises where many simulation runs must be completed with different parameters to obtain the comparative results. A Type 1 pipeline is illustrated in the *space-time diagram* shown in Figure 5.4. In this diagram, each process is assumed to have been given the same time to complete its task. Each time period is one *pipeline cycle*. Each instance of the problem requires six sequential processes, P_0, P_1, P_2, P_3, P_4, and P_5. Note the staircase effect at the beginning. After the staircase effect, one instance of the problem is completed in each pipeline cycle. The same information shown in Figure 5.4 is shown in an alternative space-time diagram in Figure 5.5, where the instances are listed along the vertical axis. This form of diagram is sometimes useful if it is necessary to show information passing from one task instance to another (as would occur in processor pipelines).

With p processes constituting the pipeline and m instances of the problem to execute, the number of pipeline cycles to execute all m instances is given by $m + p - 1$ cycles. The average number of cycles is $(m + p - 1)/m$ cycles, which tends to one cycle per problem instance for large m. In any event, one instance of the problem will be completed in each pipeline cycle after the first $p - 1$ cycles (the *pipeline latency*). The formula $m + p - 1$ for a p-stage pipeline computing m instances of a problem will be used in our analyses later.

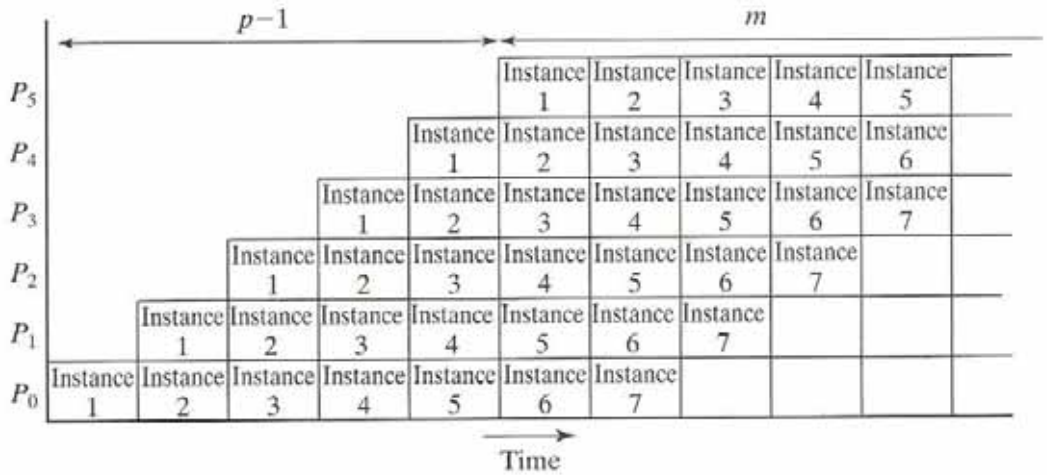


Figure 5.4 Space-time diagram of a pipeline.

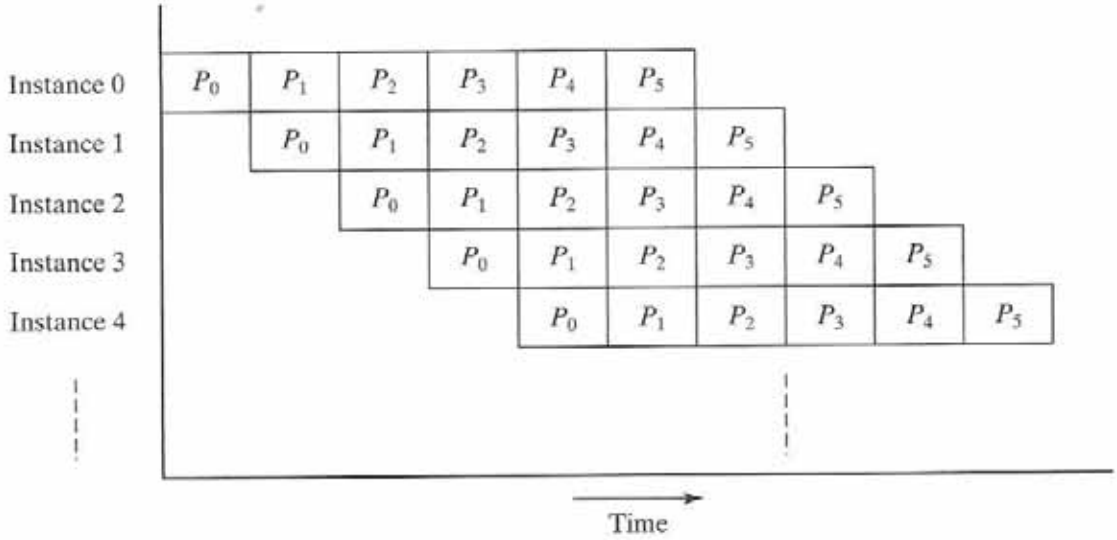
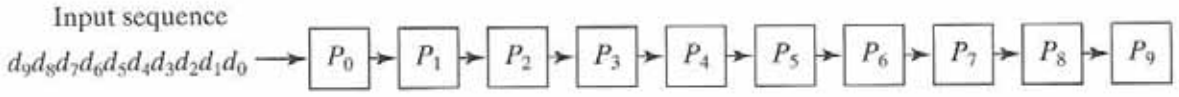
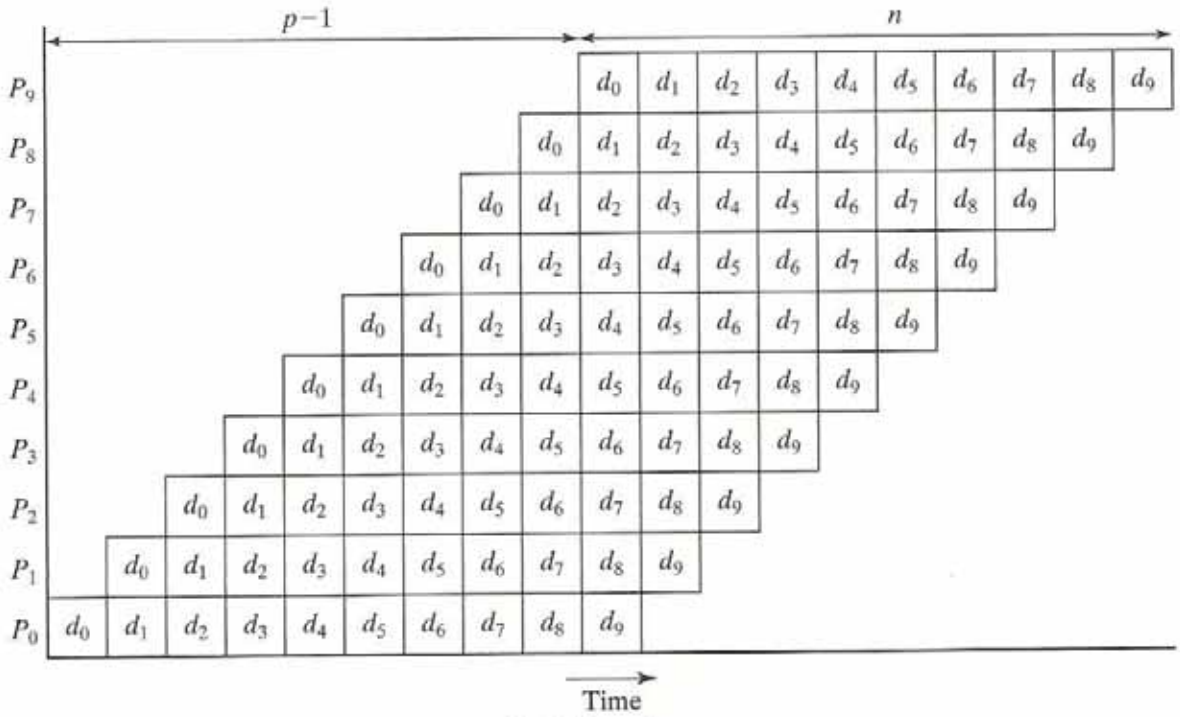


Figure 5.5 Alternative space-time diagram.

The Type 2 arrangement, in which a series of data items must be processed in a sequence, appears in arithmetic calculations, such as multiplying elements of an array where individual elements enter the pipeline as a sequential series of numbers. The arrangement is shown in Figure 5.6, where in this case ten processes form the pipeline and ten



(a) Pipeline structure



(b) Timing diagram

Figure 5.6 Pipeline processing ten data elements.

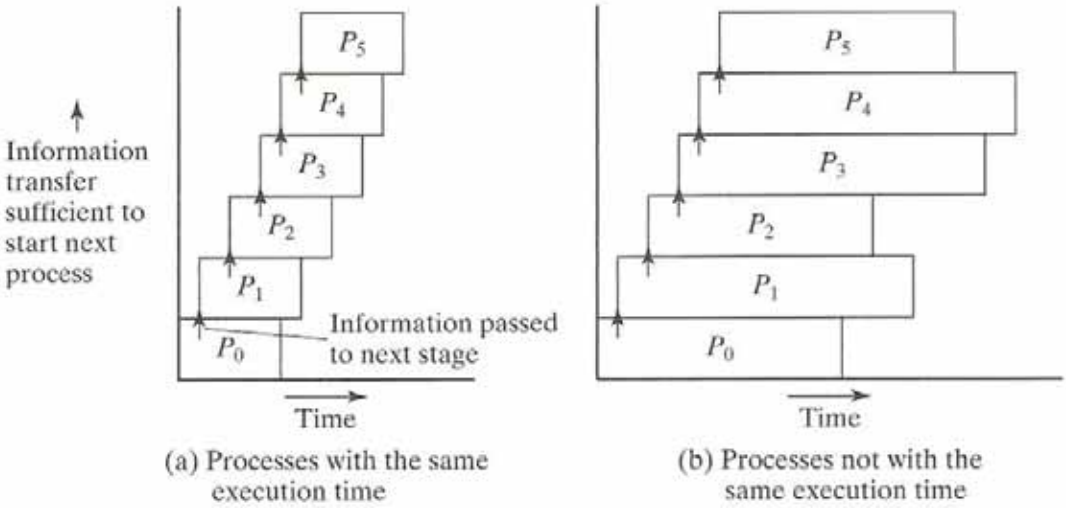


Figure 5.7 Pipeline processing where information passes to next stage before end of process.

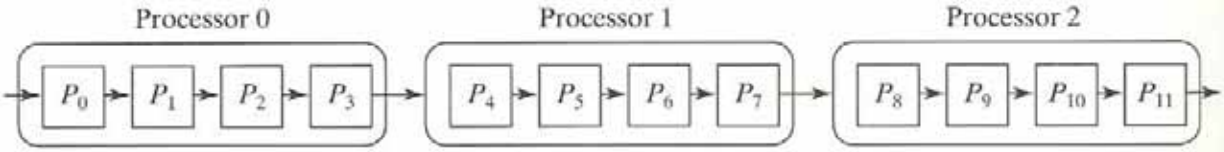


Figure 5.8 Partitioning processes onto processors.

elements, d_0, d_1, d_2, d_3, d_4, d_5, d_6, d_7, d_8, and d_9, are being processed. With p processes and n data items, the overall execution time is again given by $(p - 1) + n$ pipeline cycles, assuming these are all equal.

It is often the third arrangement, Type 3, that is utilized in parallel programs where there is only one instance of the problem to execute, but each process can pass on information to the next process before it has completed. Figure 5.7 shows space-time diagrams when information can pass from one process to another before the end of the execution of a process.

If the number of stages is larger than the number of processors in any pipeline, a group of stages can be assigned to each processor, as shown in Figure 5.8. Of course, now the pipeline stages within one processor are executed sequentially.

5.2 COMPUTING PLATFORM FOR PIPELINED APPLICATIONS

A key requirement for pipelining is the ability to send messages between adjacent processes in the pipeline. This suggests direct communication links between the processors onto which adjacent processes are mapped. An ideal interconnection structure is a line or ring structure, such as a line of processors connected to a host system, as shown in Figure 5.9. Lines and rings can be embedded into meshes and hypercubes, thereby making them suitable platforms. The seemingly inflexible line configuration is, in fact, very convenient for many applications, yet at very low cost. To use pipelining with networked computers and computer clusters efficiently requires an interconnection structure that can provide

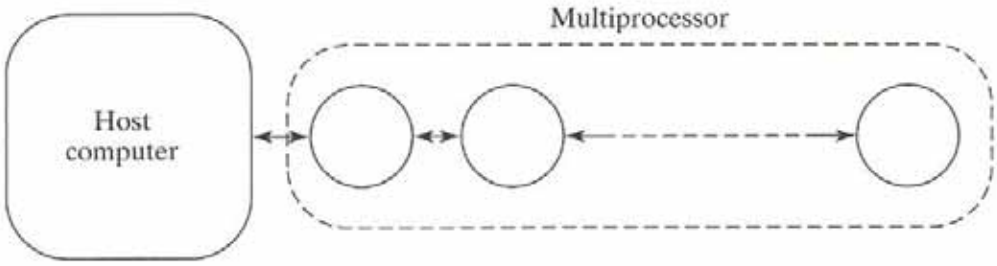


Figure 5.9 Multiprocessor system with a line configuration.

simultaneous transfers between adjacent processes or processors. Most computer clusters employ a switched interconnection structure that allows such transfers. A single shared Ethernet connection would not provide such simultaneous transfers. A little flexibility can be achieved on this matter by using (locally) blocking `send()`s (the `send()`s that are normally used). Then a process can continue with the next operation without waiting for the destination to be ready.

5.3 PIPELINE PROGRAM EXAMPLES

In this section we will examine sample problems that can be pipelined and classify the solutions as Type 1, Type 2, or Type 3.

5.3.1 Adding Numbers

For our first example, consider the problem of adding a list of numbers. (The problem could use any associative operation on a sequence of numbers.) A pipeline solution could have each process in the pipeline add one number to an accumulating sum, as shown in Figure 5.10, when one number is held in each process ($p = n$). The partial sum is passed from one process to the next, each process adding its number to the accumulating sum.

The basic code for process P_i is simply

```
recv(&accumulation, P_{i-1});
accumulation = accumulation + number;
send(&accumulation, P_{i+1});
```

except for the first process, P_0, which is

```
send(&number, P_1);
```

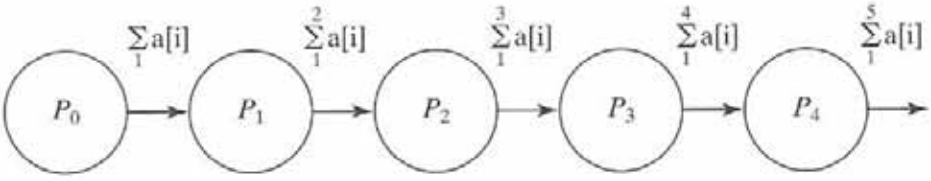


Figure 5.10 Pipelined addition.

and the last process, P_{p-1}, which is

```
recv(&number, P_{p-2});
accumulation = accumulation + number;
```

Thus, an SPMD program could have the form

```
if (process > 0) {
   recv(&accumulation, P_{i-1});
   accumulation = accumulation + number;
}
if (process < p-1) send(&accumulation, P_{i+1});
```

The final result is in the last process. Instead of addition, other arithmetic operations could be done. For example, a single number, x, could be raised to a power by multiplying the input number by x and passing on the result. Hence, a five-stage pipeline could be used to obtain x^6. Problem 5-1 explores the advantages of this approach compared to using a divide-and-conquer tree structure for the same computation.

In our general description of pipelines, we show the data being entered into the first process rather than already in the appropriate processes. If the input data is entered into the first process, it would be natural to return the result through the last process, as shown in Figure 5.11. This would be particularly appropriate if the processors were connected in a ring architecture.

For a master-slave organization, the organization shown in Figure 5.12 would also be appropriate. The numbers of one problem are entered into each process when they are needed by the processes. The first process gets its number before the others. The second process gets its number one cycle later, and so on. As we will see in Chapter 11, this form of message-passing appears in several numeric problems. In Chapter 11, we will also see pipelines where the information is entered from both ends simultaneously and two-dimensional pipeline structures.

Coming back to our problem of adding numbers, it does not make sense to have one process for each number because then to add, say, 1000 numbers, 1000 slave processes would be needed. Generally, a group of numbers would be added together in each process and the result passed onward. Such *data partitioning* is used in most numeric applications to reduce the communication overhead and is assumed in all our examples.

Analysis. Our first pipeline example is Type 1; it is efficient only if we have more than one instance of the problem to solve (i.e., more than one set of numbers to add together).

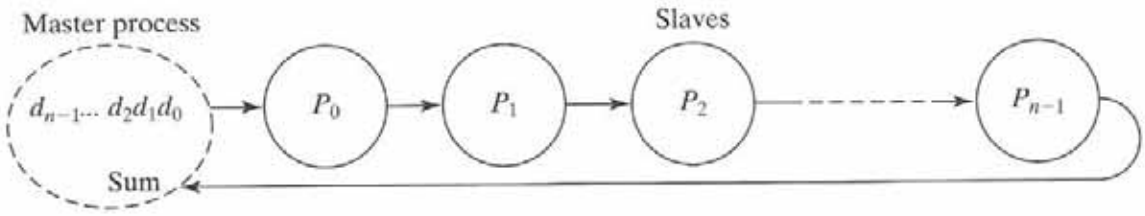


Figure 5.11 Pipelined addition numbers with a master process and ring configuration.

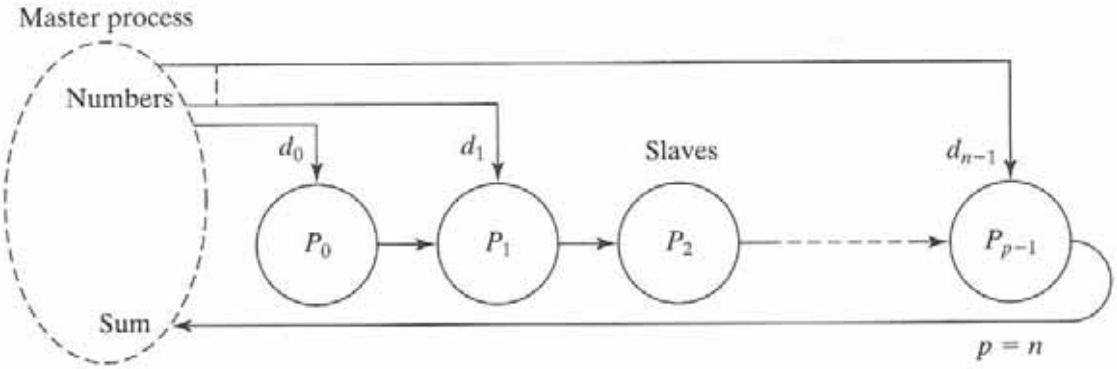


Figure 5.12 Pipelined addition of numbers with direct access to slave processes.

For analyses of pipelines, it may not be appropriate to consider all the processes as having simultaneous communication and computation phases, as in previous chapters, because each instance of a problem starts at a different time and ends at a different time. Instead, we will assume that each process performs similar actions in each pipeline cycle. Then we will work out the computation and communication required in a pipeline cycle. The total execution time, t_{total}, is then obtained by using our pipeline formula for the number of cycles multiplied by the time of one cycle; that is,

$$t_{total} = (\text{time for one pipeline cycle})(\text{number of cycles})$$

$$t_{total} = (t_{comp} + t_{comm})(m + p - 1)$$

where there are m instances of the problem and p pipeline stages (processes). The computation and communication times are given by t_{comp} and t_{comm}, respectively. The average time for a computation is given by

$$t_a = \frac{t_{total}}{m}$$

Let us assume that the structure shown in Figure 5.11 is used, and there are n numbers.

Single Instance of Problem. Let us first consider a single number being added in each stage; that is, n is equal to p. The period of one pipeline cycle will be dictated by the time of one addition and two communications, one communication from the left and one communication to the right,

$$t_{comp} = 1$$

$$t_{comm} = 2(t_{startup} + t_{data})$$

and each pipeline cycle, t_{cycle}, requires at least $t_{comp} + t_{comm}$.

$$t_{cycle} = 2(t_{startup} + t_{data}) + 1$$

The term t_{data} is the usual time to transfer one data word, and $t_{startup}$ is the communication startup time. The last process only has one communication, but this may not help because all the processes are allocated the same time period.

If we were to have only one set of numbers ($m = 1$), the total execution time, t_{total}, would be given by

$$t_{total} = (2(t_{startup} + t_{data}) + 1)n$$

(i.e., n pipeline cycles, because each process must wait for the preceding process to complete its computation and pass on its results). The time complexity is $O(n)$.

Multiple Instances of Problem. If, however, we have m groups of n numbers to add, each group resulting in a separate answer, the time of one pipeline cycle remains the same, but now we have $m + n - 1$ cycles, leading to

$$t_{total} = (2(t_{startup} + t_{data}) + 1)(m + n - 1)$$

For large m, the average execution time, t_a, is approximately

$$t_a = \frac{t_{total}}{m} \approx 2(t_{startup} + t_{data}) + 1$$

that is, one pipeline cycle.

Data Partitioning with Multiple Instances of Problem. Now let us consider data partitioning with each stage processing a group of d numbers. The number of processes is given by $p = n/d$. Each communication will still transfer one result, but the computation will also now require d numbers to be accumulated ($d - 1$ steps) plus the incoming number, so that the following applies:

$$t_{comp} = d$$

$$t_{comm} = 2(t_{startup} + t_{data})$$

$$t_{total} = (2(t_{startup} + t_{data}) + d)(m + n/d - 1)$$

Obviously, as we increase d, the data partition, the impact of the communication on the overall time diminishes. But increasing the data partition decreases the parallelism and often increases the execution time. It is left as an exercise to explore the trade-offs of these effects (Problem 5-5).

5.3.2 Sorting Numbers

The objective of sorting numbers is to reorder a set of numbers in increasing (or decreasing) numeric order (strictly, nondecreasing/nonincreasing order if there are duplicate numbers). A pipeline solution for sorting is to have the first process, P_0, accept the series of numbers one at a time, store the largest number so far received, and pass onward all numbers smaller than the stored number. If a number received is larger than the currently stored number, it replaces the currently stored number and the original stored number is passed onward. Each subsequent process performs the same algorithm, storing the largest number so far received. When no more numbers are to be processed, P_0 will have the largest number, P_1 the next-largest number, P_2 the next-largest number, and so on. This algorithm is a parallel version of *insertion sort*. The sequential version is akin to placing playing cards in order by moving cards over to insert a card in position (see Cormen, Leiserson, and Rivest, 1990). (The sequential insertion sort algorithm is only efficient for sorting a small quantity of numbers.)

Figure 5.13 shows the actions in sorting five numbers. The basic algorithm for process P_i is

```
recv(&number, P_{i-1});
if (number > x) {
   send(&x, P_{i+1});
   x = number;
} else send(&number, P_{i+1});
```

With n numbers, how many the ith process is to accept is known; it is given by $n - i$. How many to pass onward is also known; it is given by $n - i - 1$, since one of the numbers received is not passed onward. Hence, a simple loop could be used:

```
right_procNum = n - i - 1;          /* number of processes to the right */
recv(&x, P_{i-1});
for (j = 0; j < right_procNum; j++) {
   recv(&number, P_{i-1});
   if (&number > x) {
      send(&x, P_{i+1});
      x = number;
   } else send(&number, P_{i+1});
}
```

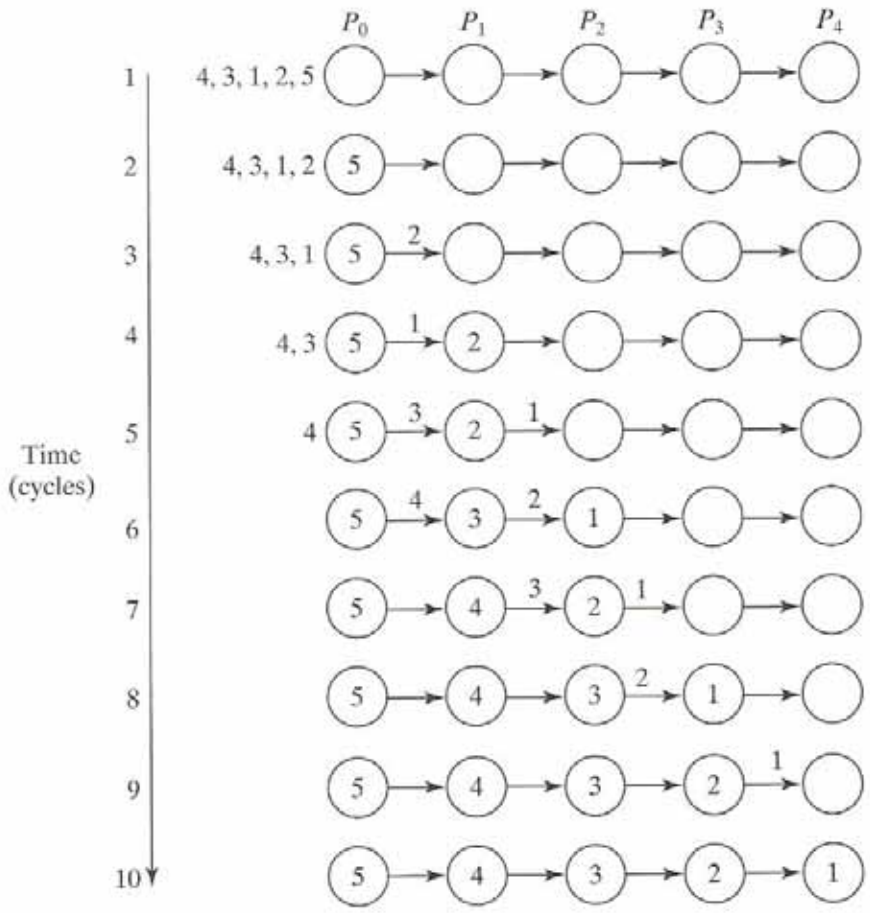


Figure 5.13 Steps in insertion sort with five numbers.

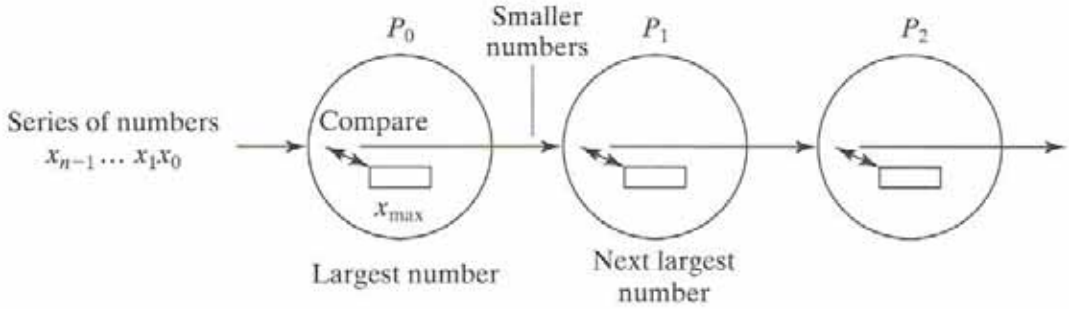


Figure 5.14 Pipeline for sorting using insertion sort.

The pipeline is illustrated in Figure 5.14. A message-passing program using an SPMD or a master-slave approach is straightforward, especially since each pipeline process executes essentially the same code. We see from the code that there is virtually no opportunity for a process to continue with useful work in one pipeline cycle after passing on a number (Type 3). However, a series of operations is performed on a series of data items (Type 2), and this leads to a significant speedup even on one instance of the problem.

Results of the sorting algorithm can be extracted from the pipeline using either the ring configuration of Figure 5.11 or the bidirectional line configuration of Figure 5.15. The latter is advantageous because a process can return its result as soon as the last number is passed through it. The process does not have to wait for all the numbers to be sorted. Leighton (1992) describes this and three other ways that the results could be collected, but concludes that the way described here is best.

Incorporating results being returned, process i could have the form

```
right_procNum = n - i - 1;          /* number of processes to the right */
recv(&x, P_{i-1});
for (j = 0; j < right_procNum; j++) {
   recv(&number, P_{i-1});
   if (number > x) {
      send(&x, P_{i+1});
      x = number;
   } else send(&number, P_{i+1});
}
send(&x, P_{i-1});                                  /* send number held */
for (j = 0; j < right_procNum; j++) {               /* pass on other numbers */
   recv(&number, P_{i+1});
   send(&number, P_{i-1});
}
```

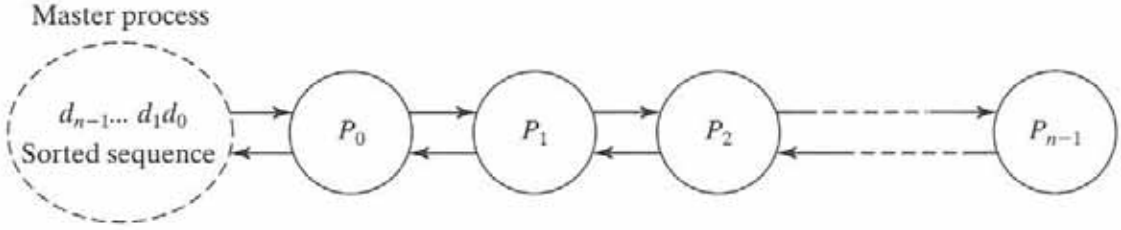


Figure 5.15 Insertion sort with results returned to the master process using a bidirectional line configuration.

Now more numbers pass through a process as processes get nearer the master process. Great care is needed in programming to ensure that the correct number of `send()`s and `recv()`s are present in each process. Process $n-1$ has no `recv()`s and one `send()` (to process $n-2$). Process $n-2$ has one `recv()` (from process $n-1$) and two `send()`s (to process $n-3$), and so on. It would be very easy to mismatch `send()`s and `recv()`s, in which case a deadlock situation would occur.

Analysis. Assuming that the compare-and-exchange operation is regarded as one computational step, a sequential implementation of the algorithm requires

$$t_s = (n-1) + (n-2) + \ldots + 2 + 1 = \frac{n(n-1)}{2}$$

as it takes $n-1$ steps to find the largest number, $n-2$ steps to find the next-largest number using the remaining numbers, and so on. The approximate number of steps is $n^2/2$, obviously a very poor sequential sorting algorithm and unsuitable except for very small n.

The parallel implementation has $n + n - 1 = 2n - 1$ pipeline cycles during the sorting if there are n pipeline processes and n numbers to sort. Each cycle has one compare and exchange operation. Communication consists of one `recv()` and one `send()`, except for the last process, which only has a `recv()`; this minor difference can be ignored. Therefore, each pipeline cycle requires at least

$$t_{comp} = 1$$

$$t_{comm} = 2(t_{startup} + t_{data})$$

The total execution time, t_{total}, is given by

$$t_{total} = (t_{comp} + t_{comm})(2n-1) = (1 + 2(t_{startup} + t_{data}))(2n-1)$$

If the results are returned by communication to the left through to the master, a timing diagram is obtained, as shown in Figure 5.16, leading to $3n - 1$ pipeline cycles (Leighton, 1992). Of course, a real parallel program does not operate completely in synchronism, as suggested in these diagrams, because of the delays in the communication media and other variations.

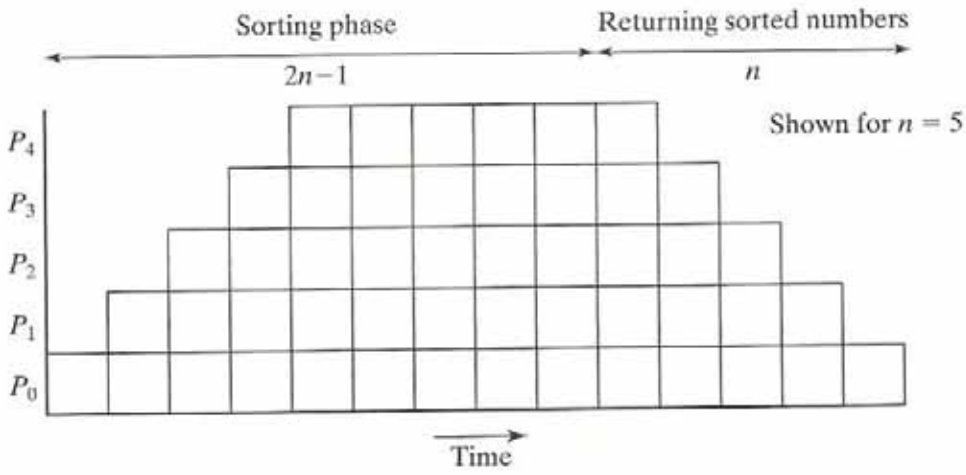


Figure 5.16 Insertion sort with results returned.

5.3.3 Prime Number Generation

A classical method of extracting prime numbers is the sieve of Eratosthenes, described by Eratosthenes of Cyrene more than two thousand years ago (Bokhari, 1987). In this method, a series of all integers is generated from 2. The first number, 2, is prime and kept. All multiples of this number are deleted because they cannot be prime. The process is repeated with each remaining number. The algorithm removes nonprimes, leaving only primes.

For example, suppose we want the prime numbers from 2 to 20. We start with all the numbers:

$$2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20$$

After considering 2, we get

$$2, 3, \cancel{4}, 5, \cancel{6}, 7, \cancel{8}, 9, \cancel{10}, 11, \cancel{12}, 13, \cancel{14}, 15, \cancel{16}, 17, \cancel{18}, 19, \cancel{20}$$

where the numbers with / are marked as not prime and not to be considered further. After considering 3, we get

$$2, 3, \cancel{4}, 5, \cancel{6}, 7, \cancel{8}, \cancel{9}, \cancel{10}, 11, \cancel{12}, 13, \cancel{14}, \cancel{15}, \cancel{16}, 17, \cancel{18}, 19, \cancel{20}$$

Subsequent numbers are considered in a similar fashion. However, to find the primes up to n, it is only necessary to start at numbers up to $\sqrt{n}$. All multiples of numbers greater than $\sqrt{n}$ will have been removed because they are also a multiple of some number equal to or less than $\sqrt{n}$. For example, if $n = 256$, ($\sqrt{n} = 16$), it is not necessary to consider multiples of the number 17 because 17×2 will have been removed as 2×17, 17×3 will have been removed as 3×17, and so on for other numbers beyond 16. Therefore in our example we have all the primes up to 20 by using 2 and 3.

We should mention in passing that the basic method as described is not suitable for finding very large primes (which are of most interest) sequentially because the sequential time complexity is significant and is dominated by the early passes through the list. A simple way to improve the performance is to consider odd numbers only (a way which is left as an exercise).

Sequential Code. A sequential program for this problem usually employs an array with elements initialized to 1 (TRUE) and set to 0 (FALSE) when the index of the element is not a prime number. Letting the square root of `n` be `sqrt_n`, we might have

```
for (i = 2; i <= n; i++)
   prime[i] = 1;                          /* Initialize array */
for (i = 2; i <= sqrt_n; i++)             /* for each number */
   if (prime[i] == 1)                     /* identified as prime */
      for (j = i + i; j <= n; j = j + i)  /* strike out all multiples */
         prime[j] = 0;                    /* includes already done */
```

The elements in the array still set to 1 identify the primes (given by the array indices). Then, the primes are found by examining the array for 1s.

The number of iterations striking out multiples of primes will depend upon the prime. There are $\lfloor n/2 - 1 \rfloor$ multiples of 2, $\lfloor n/3 - 1 \rfloor$ multiples of 3, and so on. Hence, the total

sequential time is given by

$$t_s = \left\lfloor \frac{n}{2} - 1 \right\rfloor + \left\lfloor \frac{n}{3} - 1 \right\rfloor + \left\lfloor \frac{n}{5} - 1 \right\rfloor + \ldots + \left\lfloor \frac{n}{\sqrt{n}} - 1 \right\rfloor$$

assuming the computation in each iteration equates to one computational step. The sequential time complexity is $O(n^2)$.

This implementation is very inefficient in that the inner loop will strike out numbers that may have already been deleted by a previous number. In fact, each sweep need only start at i^2 rather than $2i$, where i is the prime number. For example, considering multiples of 5, the sweep can start at 25 (i.e., 5×5) as 5×2, 5×3, and 5×4 will have been considered with previous prime numbers. The analysis of this version of the sieve of Eratosthenes can be found in Quinn (1994).

Parallel Code. Note that the early terms in the preceding expression will dominate the overall time. (There are more multiples of 2 than 3, more multiples of 3 than 4, and so on.) A parallel implementation based upon partitioning, where each process strikes out multiples of one number, will not be very effective. In fact, Quinn (1994) shows that the maximum speedup using this method is limited to about 2.83 irrespective of the number of processors (using certain assumptions). Bohkari (1987) also finds that this method can only use a limited number of processors in a practical situation. There are other ways this problem can be tackled. For example, each process could be assigned a range of numbers and strike out multiples in that range (see Problem 5-11).

A pipelined implementation can be quite effective. First, a series of consecutive numbers is generated that feeds into the first pipeline stage. This stage extracts all multiples of 2 and passes the remaining numbers onto the second stage. The second stage extracts all multiples of 3 and passes the remaining numbers onto the next stage, and so on. The implementation is illustrated in Figure 5.17. There have to be as many stages in the pipeline as prime numbers (unless a "block" partition is used, in which each process handles a group of numbers in the list). The pipeline implementation does not have the disadvantage of reconsidering numbers already identified as prime, as does the simple sequential version.

The code for a process, P_i, could be based upon

```
recv(&x, P_{i-1});
/* repeat following for each number */
recv(&number, P_{i-1});
if ((number % x) != 0) send(&number, P_{i+1});
```

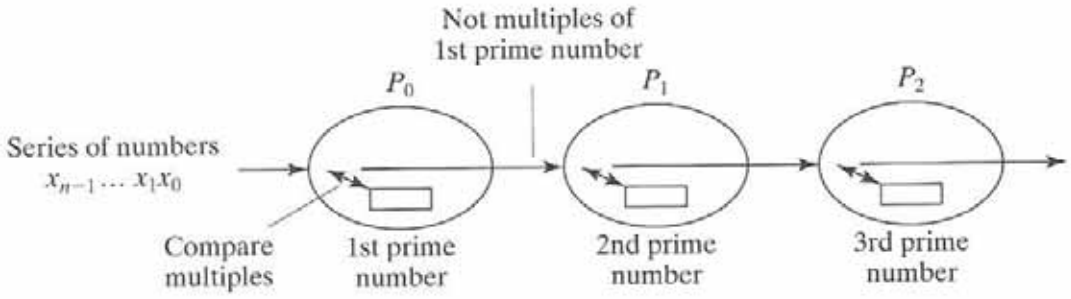


Figure 5.17 Pipeline for sieve of Eratosthenes.

A simple `for` loop is not sufficient for repeating the actions because each process will not receive the same amount of numbers and the amount is not known beforehand. A general technique for dealing with this situation in pipelines is to use a "terminator" message, which is sent at the end of the sequence. Then each process could be

```
recv(&x, P_{i-1});
for (i = 0; i < n; i++) {
   recv(&number, P_{i-1});
   if (number == terminator) break;
   if (number % x) != 0) send(&number, P_{i+1});
}
```

Note. The use of the mod operator, %, to detect whether a number is a multiple of another number is expensive (in execution time). We intentionally avoided its use in the sequential code. Avoiding its use in the parallel code is left as an exercise (Problem 5-7).

Analysis. As with the sorting example, the pipeline implementation is a Type 2. Analysis of the algorithm is similar to the sorting algorithm except that each process in the pipeline will complete fewer steps than the preceding process because it will not receive all the numbers that the preceding process receives.

5.3.4 Solving a System of Linear Equations — Special Case

The final example is Type 3, in which the process can continue with useful work after passing on information. The objective here is to solve a system of linear equations of the so-called *upper-triangular* form:

$$
\begin{aligned}
a_{n-1,0}x_0 + a_{n-1,1}x_1 + a_{n-1,2}x_2 \quad & \cdots \quad + a_{n-1,n-1}x_{n-1} &= b_{n-1} \\
\vdots \\
a_{2,0}x_0 + a_{2,1}x_1 + a_{2,2}x_2 & &= b_2 \\
a_{1,0}x_0 + a_{1,1}x_1 & &= b_1 \\
a_{0,0}x_0 & &= b_0
\end{aligned}
$$

where the a's and b's are constants and the x's are unknowns to be found. The method used to solve for the unknowns $x_0, x_1, x_2, \ldots, x_{n-1}$ is a simple repeated "back" substitution. First, the unknown x_0 is found from the last equation:

$$x_0 = \frac{b_0}{a_{0,0}}$$

The value obtained for x_0 is substituted into the next equation to obtain x_1:

$$x_1 = \frac{b_1 - a_{1,0}x_0}{a_{1,1}}$$

The values obtained for x_1 and x_0 are substituted into the next equation to obtain x_2:

$$x_2 = \frac{b_2 - a_{2,0}x_0 - a_{2,1}x_1}{a_{2,2}}$$

and so on until all the unknowns are found.

Clearly, this algorithm can be implemented as a pipeline. The first pipeline stage computes x_0 and passes x_0 onto the second stage, which computes x_1 from x_0 and passes both x_0 and x_1 onto the next stage, which computes x_2 from x_0 and x_1, and so on, as shown in Figure 5.18. Each stage is implemented with one process. There are n processes for n equations (i.e., $p = n$). The ith process ($0 < i < p$) receives the values $x_0, x_1, x_2, \ldots, x_{i-1}$ and computes x_i from the equation

$$x_i = \frac{b_i - \sum_{j=0}^{i-1} a_{i,j}x_j}{a_{i,i}}$$

Sequential Code. Given the constants $a_{i,j}$ and b_k stored in arrays `a[][]` and `b[]`, respectively, and the values for unknowns to be stored in an array, `x[]`, the sequential code could be

```
x[0] = b[0]/a[0][0];                    /* x[0] computed separately */
for (i = 1; i < n; i++) {               /* for remaining unknowns */
   sum = 0;
   for (j = 0; j < i; j++)
      sum = sum + a[i][j]*x[j];
   x[i] = (b[i] - sum)/a[i][i];
}
```

Parallel Code. The pseudocode of process P_i ($1 < i < p$) of one pipelined version could be

```
for (j = 0; j < i; j++) {
   recv(&x[j], P_{i-1});
   send(&x[j], P_{i+1});
}
sum = 0;
for (j = 0; j < i; j++)
   sum = sum + a[i][j]*x[j];
x[i] = (b[i] - sum)/a[i][i];
send(&x[i], P_{i+1});
```

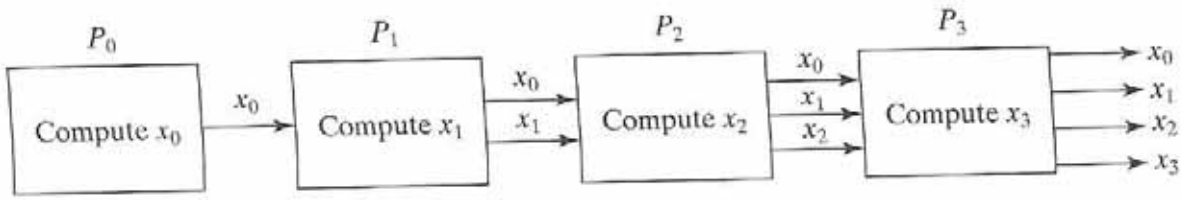


Figure 5.18 Solving an upper triangular set of linear equation using a pipeline.

P_0 simply computes x_0 and passes x_0 on. Now we have additional computations to do after receiving and resending values. This leads to a timing characteristic, as shown in Figure 5.19.[1]

The code for P_i can be written as

```
sum = 0;
for (j = 0; j < i; j++) {
   recv(&x[j], P_{i-1});
   send(&x[j], P_{i+1});
   sum = sum + a[i][j]*x[j];
}
x[i] = (b[i] - sum)/a[i][i];
send(&x[i], P_{i+1});
```

Analysis. For this pipeline, we cannot assume that the computational effort is the same at each pipeline stage (see Figure 5.19). The first process, P_0, performs one divide and one `send()`. The ith process ($0 < i < p - 1$) performs i `recv()`s, i `send()`s, i multiply/add, one divide/subtract, and a final `send()`, a total of $2i + 1$ communication times and $2i + 2$ computational steps, assuming that multiply, add, divide, and subtract are each one step. The last process, P_{p-1}, performs $p - 1$ `recv()`s, $p - 1$ multiply/add, and one divide/subtract, a total of $p - 1$ communication times and $2p - 1$ computational steps. Figure 5.20 shows the operations in which the communication time and combined multiply/add or divide/subtract are the same. In that case, we get a perfect synchronization of the `send()`s and `recv()`s, and the parallel execution time will be given by the final process plus the $p - 1$ `send()`s plus one divide (to compute x_0).

In essence, the parallel implementation has an $O(n)$ time complexity as $p = n$. The sequential version has a time complexity of $O(n^2)$. The actual speedup is not n, however. It would depend heavily on the actual system parameters. We would expect a computational step, even division, to be much faster than the time of a communication step. To reduce the overhead of communication, we have applied nonblocking send routines to allow the source process to continue as soon as possible with the next computation. The processors will, in general, be constrained by the blocking receives.

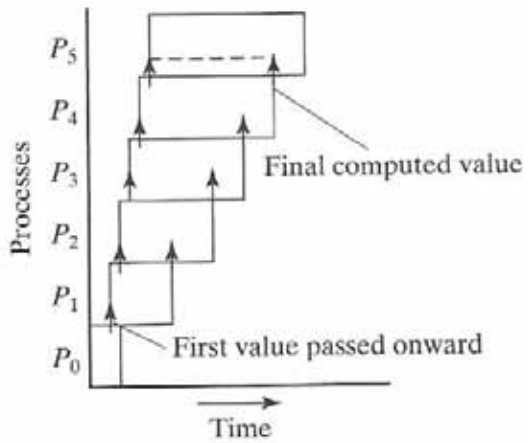


Figure 5.19 Pipeline processing using back substitution.

[1] There is another pipeline solution of implementing back substitution; see Chapter 10.

P_0	P_1	P_2	P_3	P_4
divide				
send(x_0) $\Rightarrow$	recv(x_0)			
end	send(x_0) $\Rightarrow$	recv(x_0)		
	multiply/add	send(x_0) $\Rightarrow$	recv(x_0)	
	divide/subtract	multiply/add	send(x_0) $\Rightarrow$	recv(x_0)
	send(x_1) $\Rightarrow$	recv(x_1)	multiply/add	send(x_1) $\Rightarrow$
	end	send(x_1) $\Rightarrow$	recv(x_1)	multiply/add
		multiply/add	send(x_1) $\Rightarrow$	recv(x_1)
		divide/subtract	multiply/add	send(x_1) $\Rightarrow$
		send(x_2) $\Rightarrow$	recv(x_2)	multiply/add
		end	send(x_2) $\Rightarrow$	recv(x_2)
			multiply/add	send(x_2) $\Rightarrow$
			divide/subtract	multiply/add
			send(x_3) $\Rightarrow$	recv(x_3)
			end	send(x_3) $\Rightarrow$
				multiply/add
				divide/subtract
				send(x_4) $\Rightarrow$
				end

Time ↓

Figure 5.20 Operations in back substitution pipeline.

Final Comments on Solution to Linear Equations. We have studied how to parallelize the solution of a set of linear equations having upper triangular form (which obviously also applies to a set of linear equations having lower triangular form). Such equations do occur in practice; for example, Quinn (1994) describes an upper triangular set of equations for solving for the currents in an electrical circuit consisting of resistors and voltage sources. More important, however, back substitution is an essential component of solving a general set of linear equations when using Gaussian elimination. We describe the solution of linear equations using Gaussian elimination in Chapter 10. Gaussian elimination converts a set of linear equations into triangular form, after which back substitution is used to solve the equations.

5.4 SUMMARY

This chapter introduced the following:

- The pipeline concept and its application areas
- Analysis of pipelines
- Examples illustrating the potential of pipelining, including
  - Insertion sort
  - Prime number generation
  - Solving an upper triangular system of linear equations

FURTHER READING

Pipeline processing most often is seen in specialized very large scale integration (VLSI) components designed to implement arithmetic algorithms. Apart from the simple one-dimensional pipelines with data entered at one side only, more complex pipelines or linear arrays can be devised in which the data is entered from the left and right simultaneously and information moves in both directions. Also, two-dimensional arrays can be devised, especially for implementation in VLSI. We will consider such arrays in Chapter 10 for operating on vectors and matrices. The arrays in that chapter come under the classification of systolic arrays. Pipelines can also be designed to operate upon bits of numbers to achieve various arithmetic operations. Leighton (1992) explores this use of pipelines.

The sieve of Eratosthenes is the fundamental way of finding prime numbers and has been used many times as a programming example in sequential programming texts. Bokhari (known for his early work on the multiprocessor mapping problem) describes the results of using the sieve of Eratosthenes as a benchmark program for a shared memory multiprocessor (Bokhari, 1987). Lansdowne, Cousins, and Wilkinson (1987) continue on this topic and demonstrate a way to program the sieve that improves its performance.

BIBLIOGRAPHY

BOKHARI, S. H. (1987), "Multiprocessing the Sieve of Eratosthenes," *Computer*, Vol. 20, No. 4, pp. 50–58.

CORMEN, T. H., C. E. LEISERSON, AND R. L. RIVEST (1990), *Introduction to Algorithms*, MIT Press, Cambridge, MA.

HENNESSY, J. L., AND D. A. PATTERSON (2003), *Computer Architecture: A Quantitative Approach*, 3rd edition, Morgan Kaufmann, San Mateo, CA.

LANSDOWNE, S. T., R. E. COUSINS, AND D. C. WILKINSON (1987), "Reprogramming the Sieve of Eratosthenes," *Computer*, Vol. 24, No. 8, pp. 90–91.

LEIGHTON, F. T. (1992), *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*, Morgan Kaufmann, San Mateo, CA.

QUINN, M. J. (1994), *Parallel Computing Theory and Practice*, McGraw-Hill, NY.

WILKINSON, B. (1996), *Computer Architecture Design and Performance*, 2nd edition, Prentice Hall, London.

PROBLEMS

Scientific/Numerical

5-1. Write a parallel program to compute x^{16} using a pipeline approach. Repeat by applying a divide-and-conquer approach. Compare the two methods analytically and experimentally.

5-2. Develop a pipeline solution to compute $\sin\theta$ according to

$$\sin\theta = \theta - \frac{\theta^3}{3!} + \frac{\theta^5}{5!} - \frac{\theta^7}{7!} + \frac{\theta^9}{9!} - \ldots$$

A series of values are input, $\theta_0, \theta_1, \theta_2, \theta_3, \ldots$.

5-3. Modify the program in Problem 5-2 to compute $\cos\theta$ and $\tan\theta$.

5-4. Write a parallel program using pipelining to compute the polynomial

$$f = a_0x^0 + a_1x^1 + a_2x^2 + \ldots + a_{n-1}x^{n-1}$$

to any degree, n, where the a's, x, and n are input. Compare the pipelined approach with the divide-and-conquer approach (Problem 4-8 in Chapter 4).

5-5. Explore the trade-offs of increasing the data partition in the pipeline addition described in Section 5.3.1. Write parallel programs to find the optimum data partition for your system.

5-6. Compare insertion sort (Section 5.3.2) implemented sequentially and implemented as a pipeline, in terms of speedup and time complexity.

5-7. Rework the parallel code for finding prime numbers in Section 5.3.3 to avoid the use of the mod operator to make the algorithm more efficient.

5-8. Radix sort is similar to the bucket sort described in Chapter 4, Section 4.2.1, but specifically uses the bits of the number to identify the bucket into which each number is placed. First the most significant bit is used to place each number into one of two buckets. Then the next-most significant bit is used to place each number in each bucket into one of two buckets, and so on until the least significant bit is reached. Reformulate the algorithm to become a pipeline where all the numbers are passed reordered from stage to stage until finally sorted. Write a parallel program for this method and analyze the method.

5-9. A pipeline consists of four stages, as shown in Figure 5.21. Each stage performs the operation

$$y_{out} = y_{in} + a \times x$$

Determine the overall computation performed.

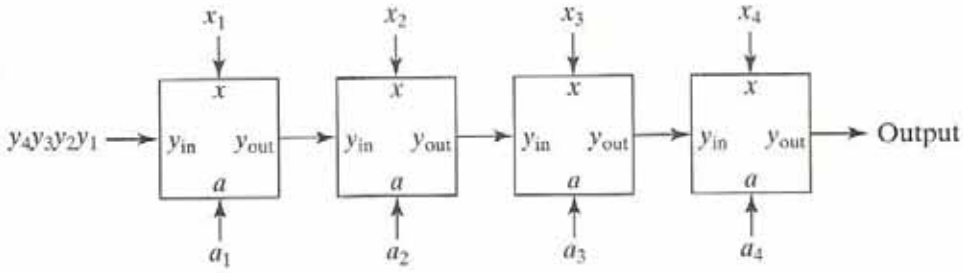


Figure 5.21 Pipeline for Problem 5-9.

5-10. The outer product of two vectors (one-dimensional arrays), A and B, produces a matrix (a two-dimensional array), C, as given by

$$AB^T = \underset{A}{\begin{bmatrix} a_0 \\ \cdot \\ \cdot \\ a_{n-1} \end{bmatrix}} \underset{B^T}{\begin{bmatrix} b_0 & \ldots & b_{n-1} \end{bmatrix}} = \underset{C}{\begin{bmatrix} a_0b_0 & \ldots & a_0b_{n-1} \\ \cdot & \ldots & \cdot \\ \cdot & \ldots & \cdot \\ a_{n-1}b_0 & \ldots & a_{n-1}b_{n-1} \end{bmatrix}}$$

Formulate pipeline implementation for this calculation given that the elements of A ($a_0, a_1, \ldots, a_{n-1}$) enter together from the left of the pipeline and one element of B is stored in one pipeline stage (P_0 stores b_0, P_1 stores b_1, etc.). Write a parallel program for this problem.

5-11. Compare implementing the sieve of Eratosthenes by each of the following ways:

(i) By the pipeline approach as described in Section 5.3.3

(ii) By having each process strike multiples of a single number

(iii) By dividing the range of numbers into m regions and assigning one region to each process to strike out multiples of prime numbers. Use a master process to broadcast each prime number as found to processes

Perform an analysis of each method.

5-12. (For those with knowledge of computer architecture.) Write a parallel program to model a five-stage RISC processor (reduced instruction set computer), as described in Hennessy and Patterson (2003). The program is to accept a list of machine instructions and shows the flow of instructions through the pipeline, including any pipeline stalls due to dependencies/resource conflicts. Use a single valid bit associated with each register to control access to registers, as described in Wilkinson (1996).

Real Life

5-13. As mentioned in Section 5.1, pipelining could be used to implement an audio frequency-amplitude histogram display in a sound system, as shown in Figure 5.22(a). This application could also be implemented by an embarrassingly parallel, functional decomposition, where each process accepts the audio input directly, as shown in Figure 5.22(b). For each method, write a parallel program to produce a frequency-amplitude histogram display using an audio file as input. Analyze both methods. (Some research may be necessary to develop how to recognize frequencies in a digitized signal.)

5-14. Due to an unprecedented rise in both automobiles and state-mandated auto inspection requirements, the citizens of the state of New Caroltucky have been complaining that it takes too long to complete the inspection process. Typically, the 35-point inspection checks for brakes (six

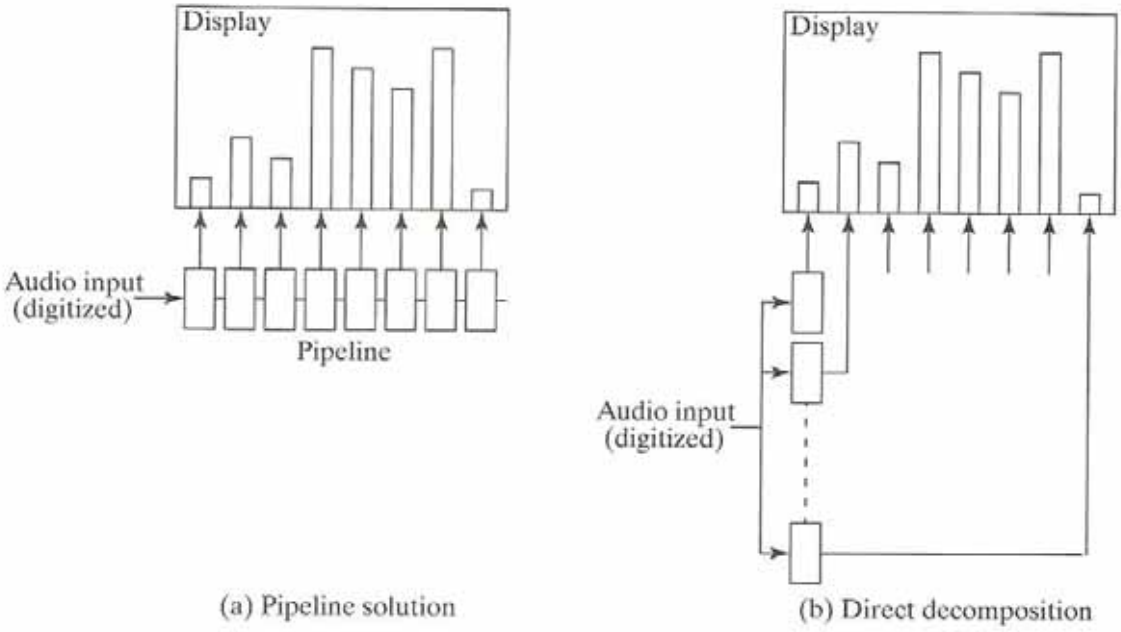


Figure 5.22 Audio histogram display.

checks there alone: each wheel is pulled and brake lining/pad thickness measured, the master cylinder integrity and performance are checked, and general brakeline leaks and cracks are looked for), along with 29 other time-consuming checks. Once a vehicle begins the inspection process, it typically takes a full hour; some claim they have had to wait in a queue just to get to the inspection bay. Legislators have been told of 72-hour queue delays in extreme cases.

The legislature of New Caroltucky is trying to decide whether the state-run inspection stations need a revamping of their operations. Since the legislature has heard that you are taking a course that includes both sequential and parallel programming concepts, it has decided to hire you to do a simulation of both the present inspection system and a proposed system. You are expected to determine the reduction in total time (queue waiting times plus inspection times) if the state revises the inspection process to implement pipelining instead of the present purely sequential approach.

The present inspection process begins with a driver entering a queue at the inspection station. When an inspector is free, the vehicle at the head of the queue is driven into the inspection bay by the inspector. The inspector then carries out each of the 35 inspections, one at a time. Assuming the vehicle passes, the inspector drives the vehicle out and puts an inspection sticker on it an hour after it was driven into the bay.

The two proposed inspection processes begin the same way, with vehicles entering a single queue. In the proposed new modified-sequential system, there will be three inspectors working in three separate bays doing inspections on three vehicles simultaneously. Each draws a vehicle from the head of the queue when ready to begin a new inspection, but sticks with that vehicle until the inspection is complete. Due to space constraints (there are only three bays), it is not possible to add more inspectors to handle more vehicles simultaneously.

In the proposed new pipelined system, the state will add some automation to the process so that a vehicle is moved automatically through the inspection bays: entering bay no. 1, moving out of it into bay no. 2 as a new vehicle is moved into bay no. 1, and moving out of it and into bay no. 3 as the second vehicle moves into bay no. 2 and a third vehicle moves into bay no. 1. Under this approach there is plenty of room to add additional inspectors in each bay to speed up the inspection steps handled in that bay. For example, if it would help, inspectors could be added for each wheel (each pulls one wheel, measures the pads/shoes, checks for wheel cylinder leaks, replaces that wheel, etc.) plus a fifth who looks for leaks in the lines. Naturally, the state is concerned about cost and efficiency; only the minimum number of inspectors required to achieve the greatest throughput are to be hired. Extra inspectors just standing around will not be tolerated unless eliminating one would cause an increase in the total inspection time once a vehicle enters the first bay.

A table of the tasks assigned to each bay, together with the times each task requires follows. In addition, the loaded labor rate for each inspector (taking into account basic salary, fringe benefits, office and paperwork costs) is given.

Your task is to simulate both new inspection systems to determine several results:

(i) What is the minimum number of inspectors needed under the proposed new pipelined system to achieve the maximum inspection throughput?

(ii) What are the labor costs per inspection performed under each proposed system?

(iii) By how much is the expected inspection delay reduced under each proposed system?

(iv) Without conducting any further simulations (analyzing only what you have obtained from this first part), give an argument for the state investing in additional facilities to expand the number of bays under both systems in order to reduce further the average inspection time. (Naturally, the tasks assigned to each bay under the pipelined approach would have to be changed, but it is assumed that the state inspectors are retrainable.)

Task table

1.	Pull left front wheel	1 minute
2.	Pull left rear wheel	1 minute
3.	Measure the pads/shoes (per wheel)	1 minute
	⋮	
i.	Replace left front wheel	1 minute
j.	Check wheel alignment	5 minutes
k.	Check exhaust system for leaks	1 minute
l.	Check engine emissions at idle	4 minutes
m.	Check engine emissions under load	3 minutes
	⋮	
z.	Remove old sticker and replace with new one	2 minutes
	Total:	60 minutes

Loaded labor rate table

1.	Line inspectors (the "worker bees")	\$40,000/yr
2.	Managers (the "drone bees")	\$60,000/yr
3.	Senior managers (the "chairman bees")	\$80,000/yr

Note 1: One manager is needed for every five (or fraction thereof) line inspectors, as well as a senior manager for every four (or fraction thereof) managers beyond the first two. For example, if there are 13 line inspectors, there would be three managers required plus one senior manager.

Note 2: This is an open-ended problem and requires the student to make some assumptions about arrival rates, randomness of arrival times, and so on, and is probably more suited to a final project in a course than simply being one of several assigned during a term.

5-15. Recall films or news reporting video in which a human chain is passing items from a stockpile area to where they are needed. (Examples include passing filled sandbags hand-to-hand up to the riverbank to build a dike to prevent the river from overflowing its banks, and a bucket-brigade in which buckets of water are being passed hand-to-hand from the water supply to the fire scene.) Given the following data, simulate an N-person chain and compare it to N persons working independently, each moving an item from the stockpile area to where it is needed. The objective is to determine the speedup, that is, the increase in the rate of delivery of the needed items, attainable through the pipelining solution versus that obtainable through the independently operating individuals' solution. Given that there are 1 million items to be moved, determine the speedup for cases in which the number of available people is 150, 300, and 3000.

Data: It is 300 meters from the stockpile to where the items are needed; a human working individually can carry one item at a time and travel at a speed of 1 meter per second carrying that item and 1.5 meters per second when traveling without an item (the return trip). Working cooperatively, the humans stand 1 meter apart and hand the item from hand-to-hand; it takes 1.25 seconds to grab an item from the person behind you, turn, and hand it to the person in front of you. Obviously, if there are only a few humans, the chain, or pipelining, approach is not practical. Similarly, if there are multiples of 300 people, multiple chains can operate in parallel.

Chapter 6

Synchronous Computations

In this chapter, we consider problems solved by a group of separate computations that must at times wait for each other before proceeding, thereby becoming synchronized. A very important class of such applications is called *fully synchronous* applications. In a fully synchronous application, all the processes are synchronized at regular points. Generally, the same computation or operation is applied to a set of data points. All the operations start at the same time in a lock-step manner analogous to SIMD computations. Seventy percent of the first set of applications studied by Fox and colleagues in the ground-breaking Caltech project were classified as synchronous applications (Fox, Williams, and Messina, 1994). First, synchronizing processes are considered and then fully synchronous applications. Finally, we describe how to reduce the amount of synchronization needed in order to increase the computational speed, which we call *partially synchronous* methods. Partially synchronous methods are very important to obtain high computational speed.

6.1 SYNCHRONIZATION

6.1.1 Barrier

Imagine a number of processes computing values. Eventually, each process must wait until all the processes have reached a particular reference point in their computations. This commonly arises when processes need to exchange data and then continue from a known state together. A mechanism is needed that prevents any process from continuing past a specified point until all the processes are ready. The basic mechanism for regulating this situation is called a *barrier*. A barrier is inserted at the point in each process where it must

wait. The processes can continue from this point when they have all reached it (or, in some implementations, when a stated number of processes have reached it.) The concept is illustrated in Figure 6.1. In this example, process P_2 is the last to reach the barrier. Therefore, all the other processes must wait and are placed in an inactive state until process P_2 reaches its barrier. Then the inactive processes are awakened (restarted) and all the processes proceed from that point.

Barriers apply to both shared memory and message-passing systems. We will discuss barriers in shared memory systems in Chapters 8 and 9. In message-passing systems, barriers are often provided with library routines. For example, MPI has the barrier routine, `MPI_Barrier()`, with a named communicator as the only parameter. `MPI_Barrier()` is called by each process in the group, blocking until every member of the group has reached the barrier call and only returning then. Although not in MPI, barriers can be defined where the number of processes that must reach the barrier to release the processes is specified and can be less than the total number of processes in the group, but using this feature would be rare. Barriers are naturally synchronous, and message tags are not used.

Figure 6.2 illustrates the library call approach for a barrier. Since a single barrier call is reused for every situation in which a barrier is required, it is essential for barriers to match with the correct barrier in other processes. This characteristic has to be ensured by the implementation. The way that the barrier call is implemented will depend upon the implementer, who in turn will be influenced by the underlying architecture. Certain underlying architectures will suggest specific efficient implementations. As usual, MPI does not specify internal implementation. However, we need to know something about the implementation to assess the complexity of the barrier. Let us review some of the common implementations of a barrier.

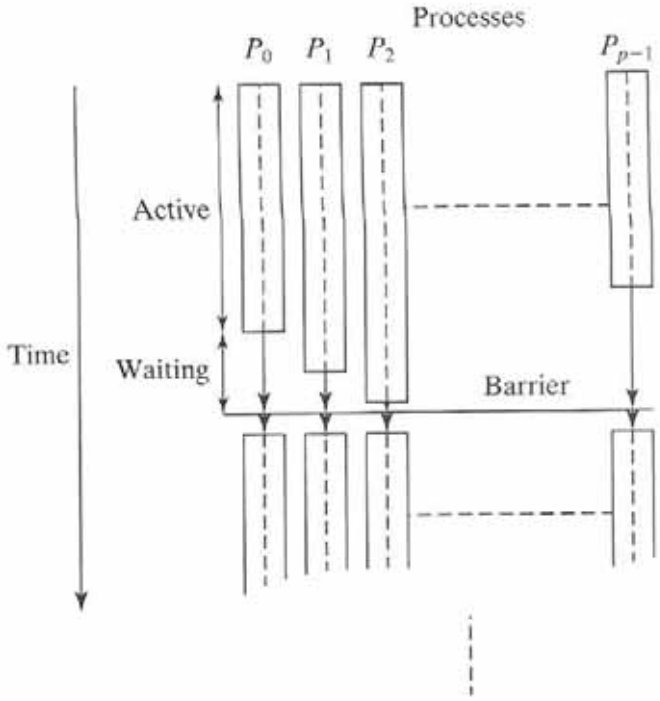


Figure 6.1 Processes reaching the barrier at different times.

wait. The processes can continue from this point when they have all reached it (or, in some implementations, when a stated number of processes have reached it.) The concept is illustrated in Figure 6.1. In this example, process P_2 is the last to reach the barrier. Therefore, all the other processes must wait and are placed in an inactive state until process P_2 reaches its barrier. Then the inactive processes are awakened (restarted) and all the processes proceed from that point.

Barriers apply to both shared memory and message-passing systems. We will discuss barriers in shared memory systems in Chapters 8 and 9. In message-passing systems, barriers are often provided with library routines. For example, MPI has the barrier routine, `MPI_Barrier()`, with a named communicator as the only parameter. `MPI_Barrier()` is called by each process in the group, blocking until every member of the group has reached the barrier call and only returning then. Although not in MPI, barriers can be defined where the number of processes that must reach the barrier to release the processes is specified and can be less than the total number of processes in the group, but using this feature would be rare. Barriers are naturally synchronous, and message tags are not used.

Figure 6.2 illustrates the library call approach for a barrier. Since a single barrier call is reused for every situation in which a barrier is required, it is essential for barriers to match with the correct barrier in other processes. This characteristic has to be ensured by the implementation. The way that the barrier call is implemented will depend upon the implementer, who in turn will be influenced by the underlying architecture. Certain underlying architectures will suggest specific efficient implementations. As usual, MPI does not specify internal implementation. However, we need to know something about the implementation to assess the complexity of the barrier. Let us review some of the common implementations of a barrier.

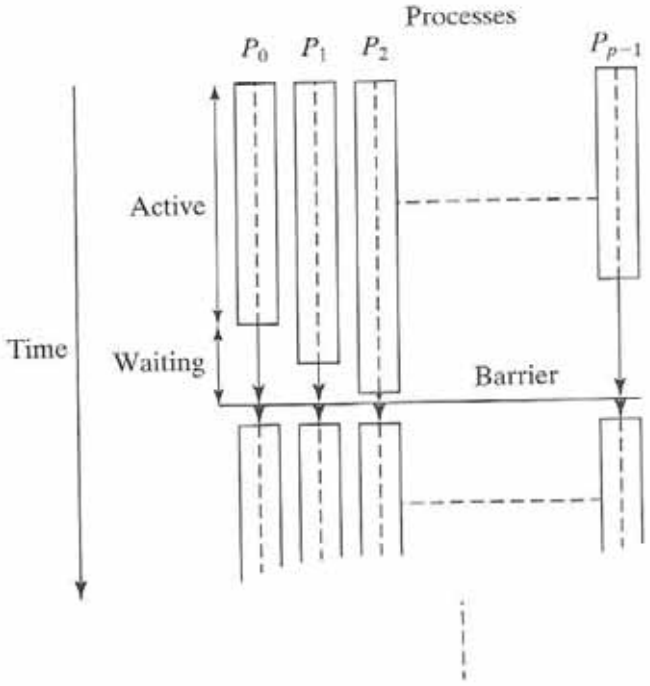


Figure 6.1 Processes reaching the barrier at different times.

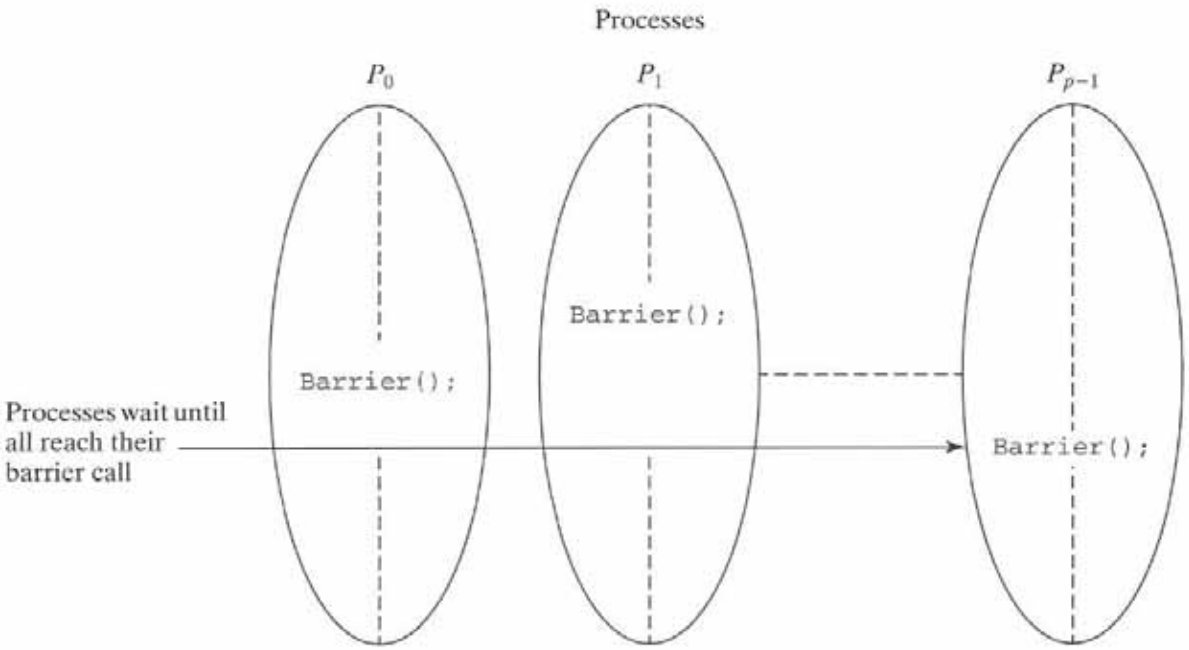


Figure 6.2 Library call barriers.

6.1.2 Counter Implementation

Figure 6.1 suggests one implementation of a barrier, a centralized counter implementation (sometimes called a *linear barrier*), as shown in Figure 6.3. A single counter is used to count the number of processes reaching the barrier. Before any process reaches its barrier, the counter is first initialized to zero. Then each process calling a barrier will increment the counter and check whether the correct number has been reached, say p. If the counter has

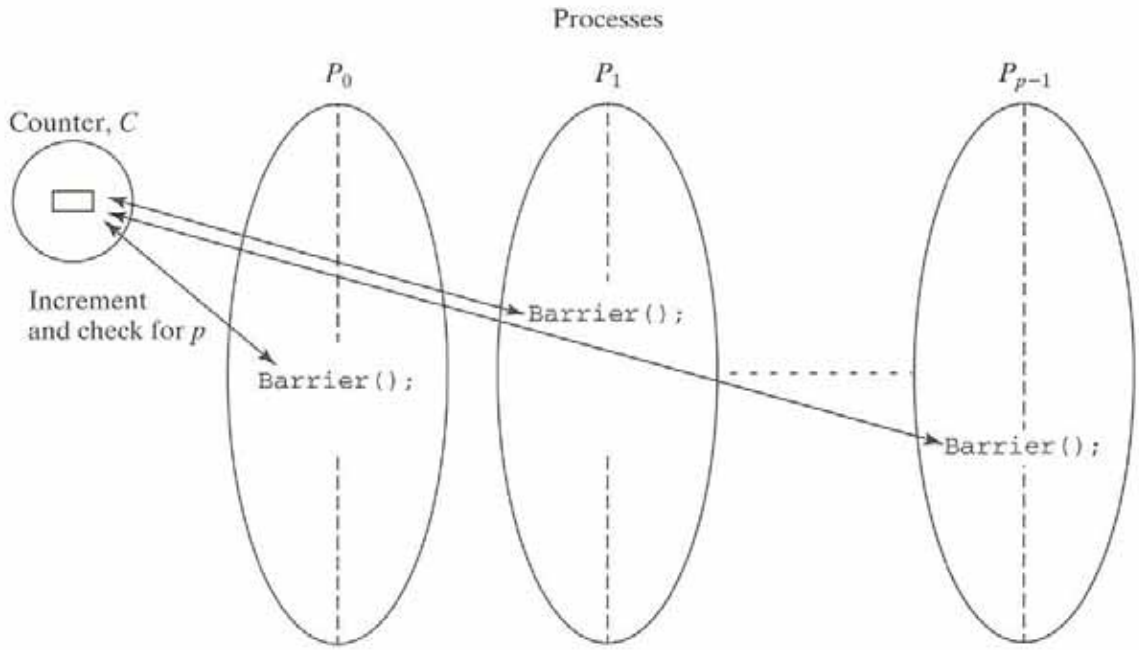


Figure 6.3 Barrier using a centralized counter.

not reached p, the process is stalled or placed in an inactive or "idle" state. If the counter has reached p, the process and all other processes waiting for the counter are released. A mechanism must be in place to release idle processes.

Counter-based barriers often have two phases, an arrival phase (or trapping) and a departure (or release) phase. A process enters the arrival phase and does not leave it until all processes have arrived in this phase. Then processes move to the departure phase and are released. Good implementations of a barrier must take into account that a barrier might be used more than once in a process. A process may enter the barrier for a second time before previous processes have left it for the first time. The two-phase design handles this scenario.

Suppose the master process maintains the barrier counter. The master process counts the messages received from "slave" processes when they reach their barrier during the arrival phase and releases slave processes in the departure phase. The code using (locally) blocking `send()`s and `recv()`s and counting using `for` loops could be of the form

```
for (i = 0; i < p; i++)     /* count slaves as they reach their barrier */
   recv(P_any);
for (i = 0; i < p; i++)     /* release slaves */
   send(P_i);
```

The variable `i` is the barrier counter. The barrier code for the slave processes is simply

```
send(P_master);
recv(P_master);
```

The complete arrangement is illustrated in Figure 6.4. Messages can be received from slave processes in any order and are accepted as received, but messages are sent to slave processes in numeric order in this code. Our implementation allows the barrier to be called repeatedly in a process because we have a clearly defined arrival phase that all processes must reach before continuing on to a clearly defined departure phase. However, note that locally blocking `send()`s do not stop the process. The slave processes will move directly to

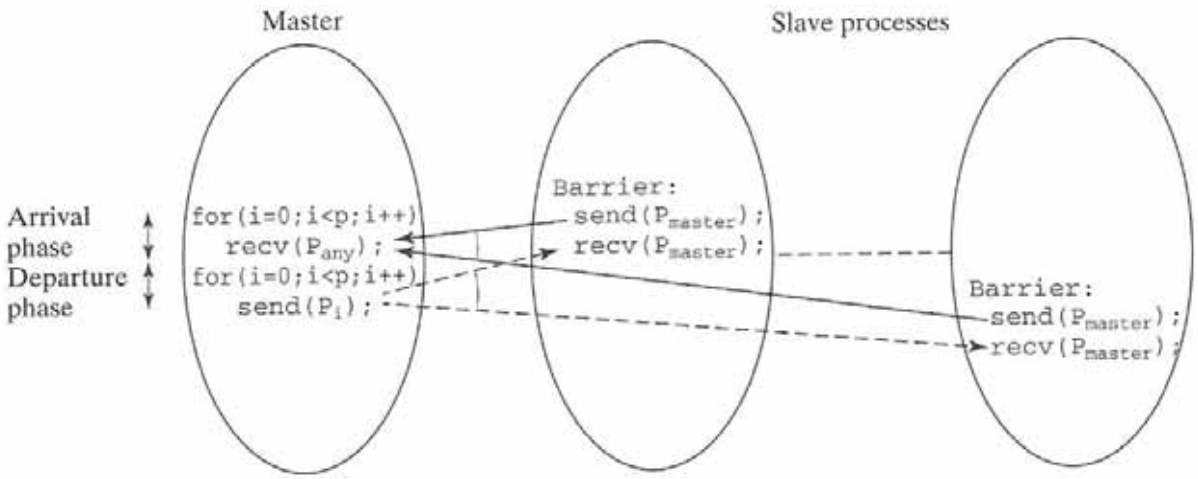


Figure 6.4 Barrier implementation in a message-passing system.

their `recv()`s after the message has been constructed for sending but before it has been received. The `recv()`s are blocking in that the processes will not move out of their departure phase until they receive their messages. The arrival phase could also be implemented with a gather routine, and the departure phase with a broadcast routine. The `send()`s and `recv()`s in Figure 6.4 do not have specific data in the message. A simple NULL message could be sent.

6.1.3 Tree Implementation

Barriers implemented with a counter have a time complexity of $O(p)$ with p processes (both the computational complexity of the master process and the communication complexity, i.e., number of messages). A more efficient barrier can be implemented using the decentralized tree construction introduced in Chapter 2 (Section 2.3.4). Suppose there are eight processes, P_0, P_1, P_2, P_3, P_4, P_5, P_6, and P_7. Essentially, the algorithm performs as follows:

First stage: P_1 sends message to P_0 (when P_1 reaches its barrier)
P_3 sends message to P_2 (when P_3 reaches its barrier)
P_5 sends message to P_4 (when P_5 reaches its barrier)
P_7 sends message to P_6 (when P_7 reaches its barrier)

Second stage: P_2 sends message to P_0 (P_2 and P_3 have reached their barrier)
P_6 sends message to P_4 (P_6 and P_7 have reached their barrier)

Third stage: P_4 sends message to P_0 (P_4, P_5, P_6, and P_7 have reached their barrier)
P_0 terminates arrival phase (when P_0 reaches barrier and has received message from P_4)

The processes now must be released from the barrier, which can be done with a reverse tree construction. The complete barrier construction is shown in Figure 6.5. In this case, the algorithm only involves sending and receiving messages without explicit computations. An eight-process algorithm with the arrival phase and departure phase both implemented with trees requires $2\log 8$ steps, or, in general, $2\log p$ steps, a communication time complexity of $O(\log p)$.

6.1.4 Butterfly Barrier

The tree construction can be developed into a so-called butterfly, in which pairs of processes synchronize at each stage in the following manner (assuming eight processes as an example):

First stage $P_0 \leftrightarrow P_1, P_2 \leftrightarrow P_3, P_4 \leftrightarrow P_5, P_6 \leftrightarrow P_7$
Second stage $P_0 \leftrightarrow P_2, P_1 \leftrightarrow P_3, P_4 \leftrightarrow P_6, P_5 \leftrightarrow P_7$
Third stage $P_0 \leftrightarrow P_4, P_1 \leftrightarrow P_5, P_2 \leftrightarrow P_6, P_3 \leftrightarrow P_7$

as shown in Figure 6.6 with two "links" between synchronizing processes, which implies two pairs of `send()/recv()`. This would be used if data were exchanged between the

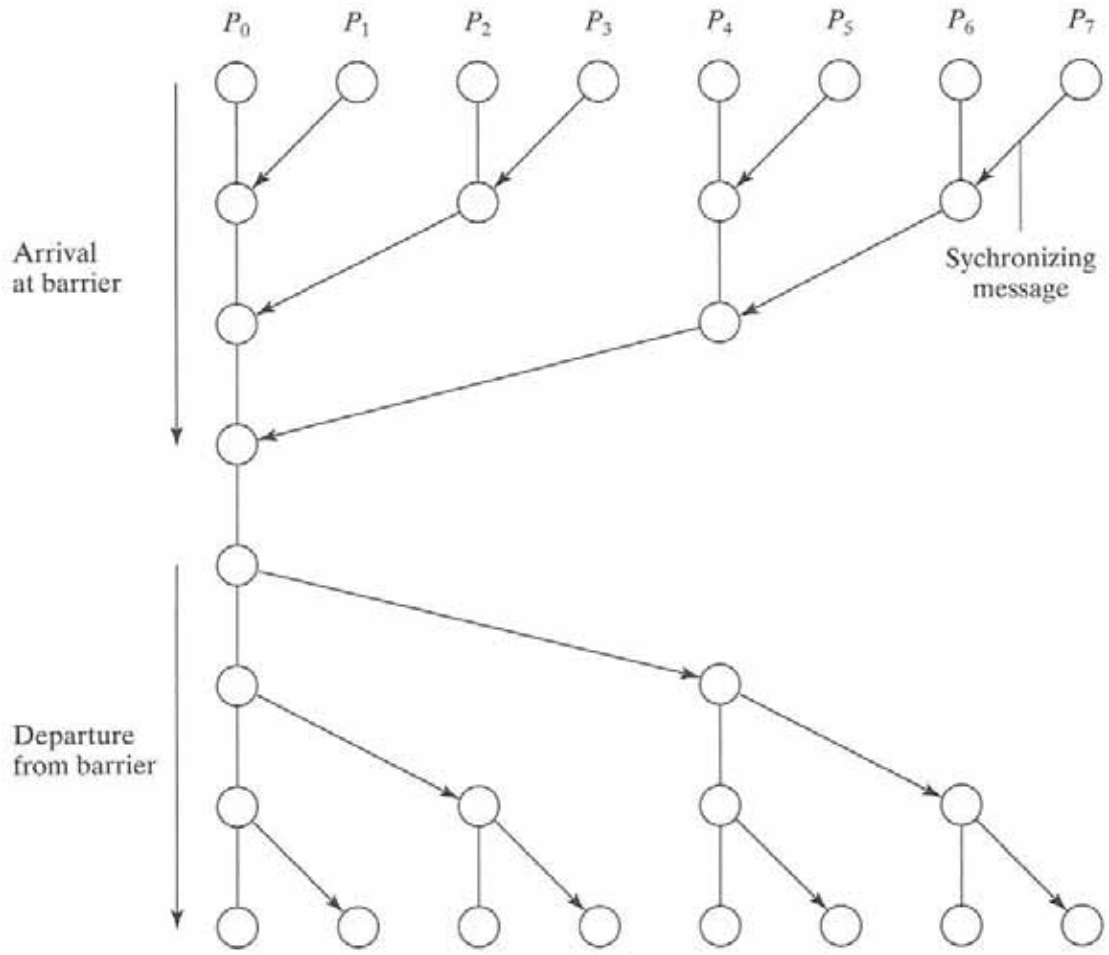


Figure 6.5 Tree barrier.

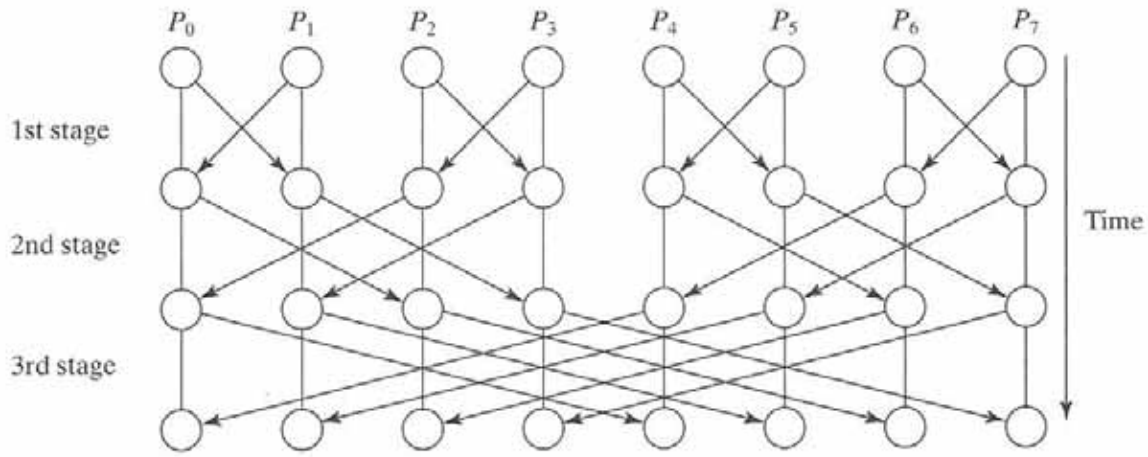


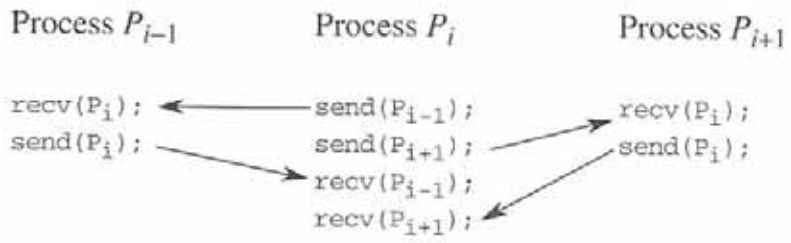
Figure 6.6 Butterfly construction.

processes (as in other applications of the butterfly). For a barrier, each synchronization requires only a single pair of `send()/recv()`. After all the synchronizing stages, each process will have been synchronized with each other process, and all processes can continue.

At stage s, process i synchronizes with process $i + 2^{s-1}$ if p is a power of 2. If p is not a power of 2, the communication is between process i and process $(i + 2^{s-1}) \bmod p$. With p processes, the butterfly has $\log p$ steps (p being a power of 2), half the number of steps of the tree implementation, but the same communication time complexity of $O(\log p)$.

6.1.5 Local Synchronization

Some problems can be formulated so that processes need only be synchronized with a few other processes, and not the complete set of processes working on the problem. This often comes about in algorithms where processes are organized as a mesh or a pipeline, and a process needs only to be synchronized to its neighbors. The message-passing technique used in Section 6.1.2 can be reduced to sending messages between individual processes that need to be synchronized. For example, suppose a process P_i needs to be synchronized and to exchange data with process P_{i-1} and process P_{i+1} before continuing. The code for this could be



Note that this is not a perfect three-process barrier because process P_{i-1} will only synchronize with P_i and continue as soon as P_i allows. Similarly, process P_{i+1} only synchronizes with P_i. However, in many applications this synchronization will be sufficient.

6.1.6 Deadlock

The tree, butterfly, and local synchronization algorithms described here employ synchronous routines to obtain the synchronization between the processes. When a pair of processes send and receive from each other, deadlock may occur. Deadlock will occur using synchronous routines (or blocking routines without sufficient buffering) if both processes perform the send first. This is because neither will return; they will wait for matching receives that are never reached. Clearly, a solution to this problem is to arrange for one process to receive first and then send and the other process to send first and then receive. In situations where even-numbered processes only communicate with odd-numbered processes, and vice versa, as in a linear pipeline, deadlock can be avoided by arranging for the even-numbered processes to perform their sends first and the odd-numbered processes to perform their receives first.

Since bidirectional data transfers are very common, a combined blocking `sendrecv()` routine can be provided in which the internal implementation details avoid deadlock. A `sendrecv()` routine sends a message to a destination process and receives a message from a source process. For flexibility, the source and destination may be different or may be the same process. MPI provides this type of routine `MPI_Sendrecv()`. It also provides `MPI_Sendrecv_replace()`, which uses a single buffer for the sending and receiving message,

replacing the send message with receive message. These routines should be implemented so that deadlock cannot occur. Applying `sendrecv()` to the preceding example would simply be

Process P_{i-1}		Process P_i		Process P_{i+1}
`sendrecv(Pi);`	↔	`sendrecv(Pi-1);`		
		`sendrecv(Pi+1);`	↔	`sendrecv(Pi);`

As a matter of detail and to give credence to those who object to the sometimes overwhelming MPI parameter list, the `MPI_Sendrecv()` routine has 12 parameters:

```
MPI_Sendrecv(sendbuf, sendcount, sendtype, dest, sendtag, recvbuf,
             recvcount, recvtype, source, recvtag, comm, status)
```

These parameters are essentially a concatenation of the parameter lists of `MPI_Send()` and `MPI_Recv()`.

6.2 SYNCHRONIZED COMPUTATIONS

6.2.1 Data Parallel Computations

A form of computation that implicitly has synchronization requirements is the *data parallel* computation. In a data parallel computation, the same operation is performed on different data elements simultaneously; that is, in parallel. Data parallel programming is very convenient for two reasons. The first is its ease of programming (essentially only one program). The second is that it can scale easily to larger problems. Many numeric and some non-numeric problems can be cast in a data parallel form. SIMD (single instruction stream multiple data stream) computers, briefly mentioned in Chapter 1, Section 1.3.4, operate as data parallel computers by having the same instruction executed by different processors but on different data, all in synchronism. In an SIMD computer, the synchronism is built into the hardware; the processors operate in lock-step fashion.

A simple example of a computation that can be formed into a data parallel computation is to add the same constant to each element of an array:

```
for (i = 0; i < n; i++)
  a[i] = a[i] + k;
```

The statement `a[i] = a[i] + k` could be executed simultaneously by multiple processors, each using a different index `i` ($0 < i < n$), as illustrated in Figure 6.7. On an SIMD computer the same instruction, equivalent to `a[] = a[] + k`, would be sent to each processor simultaneously.

A special "parallel" construct exists in parallel programming languages to specify data parallel operations — namely, the `forall` statement. The `forall` statement

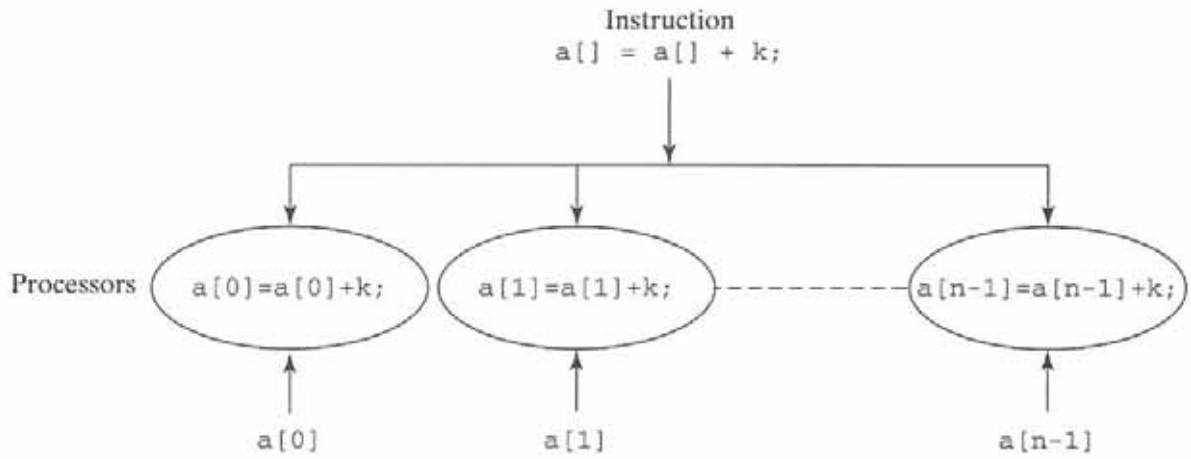


Figure 6.7 Data parallel computation.

```
forall (i = 0; i < n; i++) {
   body
}
```

states that n instances of the statements of the body can be executed simultaneously. One value of the loop variable `i` is valid in each instance of the body; the first instance has `i` = 0, the next `i` = 1, and so on. The loop variable can be used within the body to "personalize" each copy (e.g., to access different elements of an array). To illustrate this, we can add `k` to each element of an array, `a`, by writing

```
forall (i = 0; i < n; i++)
   a[i] = a[i] + k;
```

In all cases, each instance of the body must be independent of the other instances. (We explore how this can be established mathematically in Chapter 8.) The term `forall` is unfortunate, since there is no iteration here; the notation simply states that there are n copies of the body, each assigned a different value of `i`.

Although we do not consider programs for SIMD computers, the data parallel technique can be applied to multiprocessors and multicomputers. On such parallel computers, instances of the body can be executed on different processors, but the whole construct will not be completed until all instances of the body have been executed. Hence, a form of barrier is implicit within the `forall` construct. On a message-passing computer using library routines, the `forall` construct is not generally available and an explicit barrier is needed. For example, to add `k` to the elements of an array in the SPMD (single-program multiple-data) style of programming, we might write

```
i = myrank;
a[i] = a[i] + k;          /* body */
barrier(mygroup);
```

where `myrank` is a process rank between 0 and $n - 1$. It is assumed that each process has access to the required element of the array. Normally, having such a small body would not be efficient because of the barrier overhead.

We can construct much more complex SIMD computations than adding a constant to the elements of an array. Hillis and Steele (1986) describe several data parallel algorithms, including those for summing numbers, sorting, and operating on linked lists. Some SIMD computers and data parallel algorithms operate on bit patterns rather than complete numbers. Many of the image-processing algorithms described in Chapter 12 are data parallel algorithms operating upon bit patterns.

Prefix Sum Problem. An example of a data parallel algorithm is the *prefix sum* problem. In the *prefix sum* problem, given a list of numbers, $x_0, \ldots, x_{n-1}$, all the partial summations (i.e., x_0; $x_0 + x_1$; $x_0 + x_1 + x_2$; $x_0 + x_1 + x_2 + x_3$; ...) are computed. The prefix calculation can also be defined with associative operations other than addition; for example, multiplication, maximum, minimum, string concatenation, and logical (Boolean) operations (AND, OR, exclusive OR, etc.). It is widely studied in connection with various computational models. It does have practical applications in areas such as processor allocation, data compaction, sorting, and polynomial evaluation (Wagner and Han, 1986).

The sequential code for the prefix sum problem could be

```
sum[0] = x[0];
for (i = 1; i < n; i++)
   sum[i] = sum[i-1] + x[i];
```

This is an $O(n)$ algorithm.

Figure 6.8 shows a data parallel method of adding all the partial sums of 16 numbers described by Hillis and Steele (1986). This method has a multiple treelike construction and computes the partial sums in the locations $x[i]$ $(0 \le i < 16)$. The original numbers are lost. (A separate array could be used to save the numbers.) A different number of computations occurs in each step. First, 15 $(16 - 1)$ additions occur in which $x[i-1]$ is added to $x[i]$ for $1 \le i < 16$. Then 14 $(16 - 2)$ additions occur in which $x[i-2]$ is added to $x[i]$ for $2 \le i < 16$. Then 12 $(16 - 4)$ additions occur in which $x[i-4]$ is added to $x[i]$ for $4 \le i < 16$.

In general, the method requires $\log n$ steps, where there are n numbers (and n is a power of 2). In step j $(0 \le j < \log n)$, $n - 2^j$ additions occur in which $x[i - 2^j]$ is added to $x[i]$ for $2^j \le i < n$. Hence, sequential code might be written as

```
for (j = 0; j < log(n); j++)          /* at each step */
   for (i = 2^j; i < n; i++)          /* add to accumulating sum */
      x[i] = x[i] + x[i - 2^j];
```

Because SIMD computers must send the same instruction to all processors, a mechanism is provided to inhibit certain processors from executing the instruction. To indicate this, parallel code might be written as

```
for (j = 0; j < log(n); j++)          /* at each step */
   forall (i = 0; i < n; i++)         /* add to accumulating sum */
      if (i >= 2^j) x[i] = x[i] + x[i - 2^j];
```

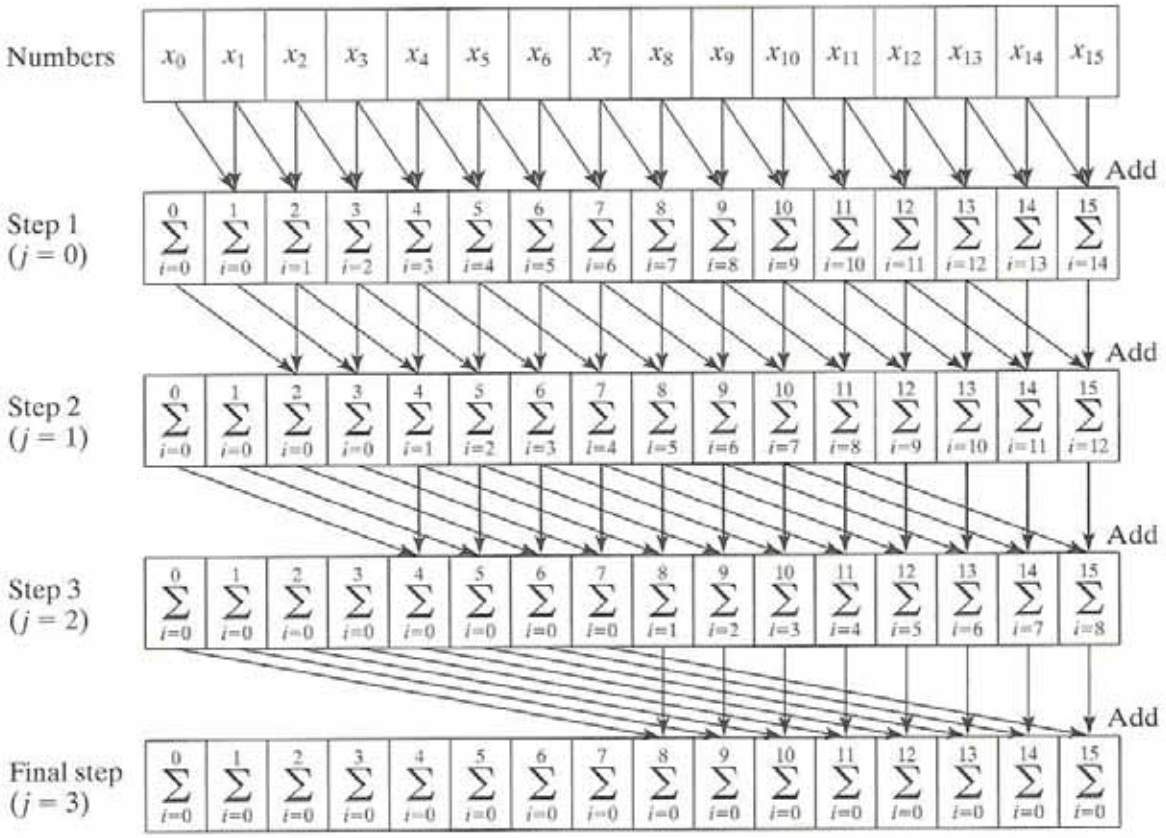


Figure 6.8 Data parallel prefix sum operation.

which uses a maximum of $n - 1$ processors and requires $\log n$ steps. The time complexity of this parallel algorithm is $O(\log n)$ in terms of both computations and communications. The efficiency is less than 1 because fewer processors are used at each step.

There is a prefix sum algorithm using a balanced tree that is also an $O(\log n)$ algorithm but requires $O(n)$ operations in total, instead of $O(n \log n)$ operations in total. See JáJá (1992) for a description of the balanced tree prefix sum algorithm.

6.2.2 Synchronous Iteration

Iteration whereby an operation is performed repeatedly is a key technique in sequential programming. Constructs are provided in all programming languages for iteration (e.g., `for`, `while`, or `do-while`). Iteration is a powerful tool for solving numerical problems, especially those which are not amenable to closed numeric solutions. Generally a result obtained on one iteration is used in the next iteration to get closer to the actual solution. The process is repeated until a sufficiently close solution is obtained. The basic idea of the iterative method is sequential in nature and appears not suited to parallel implementation. However, parallel implementation can be successfully employed to iterative methods when there are multiple independent instances of the iteration. Sometimes this is part of the problem specification. Sometimes we must rearrange the problem to obtain multiple independent instances.

The term *synchronous iteration* or *synchronous parallelism* is used to describe solving a problem by iteration where each iteration is composed of several processes that start together at the beginning of each iteration and the next iteration cannot begin until all

the processes have finished the preceding iteration. The `forall` construct could be used to specify the parallel bodies of the synchronous iteration:

```
for (j = 0; j < n; j++)                /* for each synchronous iteration */
   forall (i = 0; i < p; i++) {        /* p processes each executing */
      body(i);                         /* body using specific value of i */
   }
```

In our case for an SPMD program, we will need a specific barrier:

```
for (j = 0; j < n; j++) {              /* for each synchronous iteration */
   i = myrank;                         /* find value of i to be used */
   body(i);                            /* body using specific value of i */
   barrier(mygroup);
}
```

Let us look at some specific synchronous iteration examples.

6.3 SYNCHRONOUS ITERATION PROGRAM EXAMPLES

6.3.1 Solving a System of Linear Equations by Iteration

We saw in Chapter 5, Section 5.3.4, how to solve a system of linear equations if it was of a special triangular form. Suppose the equations were not specifically of that form, but of a general form with n equations and n unknowns

$$
\begin{array}{lll}
a_{n-1,0}x_0 + a_{n-1,1}x_1 + a_{n-1,2}x_2 \cdots & & + a_{n-1,n-1}x_{n-1} = b_{n-1} \\
 & \vdots & \\
a_{2,0}x_0 + a_{2,1}x_1 + a_{2,2}x_2 & \cdots & + a_{2,n-1}x_{n-1} = b_2 \\
a_{1,0}x_0 + a_{1,1}x_1 + a_{1,2}x_2 & \cdots & + a_{1,n-1}x_{n-1} = b_1 \\
a_{0,0}x_0 + a_{0,1}x_1 + a_{0,2}x_2 & \cdots & + a_{0,n-1}x_{n-1} = b_0
\end{array}
$$

where the unknowns are $x_0, x_1, x_2, \ldots x_{n-1}$. One way to solve these equations for the unknowns is by iteration. By rearranging the ith equation $_{(0 \le i < n)}$:

$$a_{i,0}x_0 + a_{i,1}x_1 + a_{i,2}x_2 \quad \cdots \quad + a_{i,n-1}x_{n-1} = b_i$$

to

$$x_i = (1/a_{i,i})[b_i - (a_{i,0}x_0 + a_{i,1}x_1 + a_{i,2}x_2 \ldots a_{i,i-1}x_{i-1} + a_{i,i+1}x_{i+1} \ldots + a_{i,n-1}x_{n-1})]$$

or

$$x_i = \frac{1}{a_{i,i}}\left[b_i - \sum_{j \neq i} a_{i,j}x_j\right]$$

$(0 < i < n, 0 < j < n)$. This equation gives x_i in terms of the other unknowns and can be used as an iteration formula for each of the unknowns to obtain better approximations.

The iterative method described here is called a *Jacobi iteration*. In this method, all the values of x are updated together. (Alternative methods, such as Gauss-Seidel, are described in Chapter 11.) It can be proven that the Jacobi method will converge if the diagonal values of a have an absolute value greater than the sum of the absolute values of the other a's on the row (the array of a's is *diagonally dominant*). Therefore, convergence is guaranteed if

$$\sum_{j \neq i} |a_{i,j}| < |a_{i,i}|$$

This condition is a sufficient but not a necessary condition. The method may converge even if the array is not diagonally dominant. However, the iteration formula will not work if any of the diagonal elements are zero because it would require dividing by zero.

An iterative method begins with an initial guess for all the unknowns. A possible initialization would be to set $x_i = b_i$. Then calculate new values for the unknowns using the iteration equation. These values are substituted into the iteration formulas; then the action is repeated. The iterations are continued until sufficiently accurate values for all the unknowns are obtained (assuming that the iteration formula converges). There are also "direct" methods for solving linear equations, which we discuss in Chapter 11. Iterative methods are applicable when such direct methods require excessive computations. They also have the advantage of small memory requirements, but the disadvantage is that they may not always converge.

Termination. Termination can be especially problematic in parallel formulations. We look at this topic in detail in Chapter 7 in the context of load balancing. A simple, commonly used approach is to compare the values computed in each iteration to the values obtained from the preceding iteration, and then to terminate the computation in the tth iteration when all values are within a given tolerance; that is, when

$$\left| x_i^t - x_i^{t-1} \right| < \text{error tolerance}$$

for all i, where x_i^t is the value of x_i after the tth iteration, and x_i^{t-1} is the value of x_i after the $(t-1)$th iteration. However, this does not guarantee the solution to that accuracy. Suppose the error tolerance is 1 percent and a value is computed to 1 percent of its last computed value. This is not 1 percent of the exact value of the solution. If the calculation is converging, we would expect the next computation of x_i to be less than 1 percent different from the present value, but it could be 0.9999 percent different from the present value. The next value could be 0.9998 percent different. We can see that errors might compound and the computed value could be very significantly different from the final exact value. This is illustrated in Figure 6.9 for a hypothetical problem. In addition, errors of one computed value will affect the accuracy of other computed values that use it in their calculations.

Pacheco (1997) suggests a more complex vector termination condition:

$$\sqrt{\sum_{i=0}^{n-1} (x_i^t - x_i^{t-1})^2} < \text{error tolerance}$$

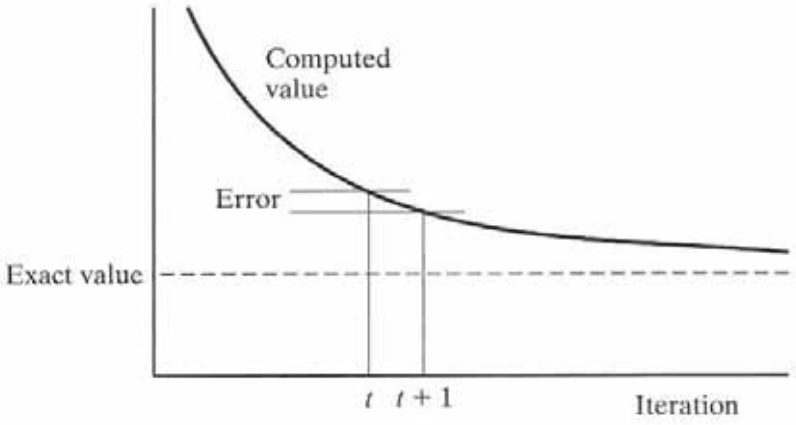


Figure 6.9 Convergence rate.

Bertsekas and Tsitsiklis (1989) suggest the termination condition

$$\left| \sum_{j=0}^{n-1} a_{i,j} x_j^t - b_i \right| < \text{error tolerance}$$

for all i. This method only uses the currently computed values and not values from the preceding iteration. For all equations, it essentially computes the left side of the equation and compares the result with the constant right side. This is not computationally intensive, since the summation without $a_{i,i}x_i$ has already been computed during the iteration. A full treatment of convergence and the effectiveness of different termination formulas can be found in texts on numerical methods.

Whatever method, since the iteration may not terminate, iterations should be stopped when a maximum number of iterations has been reached. There is a trade-off between using a complex termination calculation with potentially fewer iterations and using a less complex termination calculation with more iterations. It may be a good strategy to allow a number of iterations between checking for termination. Since the parallel formulation requires each iteration to use all the values of the preceding iteration, the calculations have to be synchronized globally. Jacobi iterations can be slow to converge. Problem 6-12 explores the convergence characteristics of Jacobi iterations empirically. Faster methods are considered in Chapter 10.

Sequential Code. Given the arrays `a[][]` and `b[]` holding the constants in the equations, `x[]` holding the unknowns, and a fixed number of iterations, the code might look like the following:

```
for (i = 0; i < n; i++)
   x[i] = b[i];                                  /*initialize unknowns*/
for (iteration = 0; iteration < limit; iteration++) {
   for (i = 0; i < n; i++) {                     /* for each unknown */
      sum = 0;
      for (j = 0; j < n; j++)          /* compute summation of a[][]x[] */
         if (i != j) sum = sum + a[i][j] * x[j];
      new_x[i] = (b[i] - sum) / a[i][i];         /* compute unknown */
   }
   for (i = 0; i < n; i++)
      x[i] = new_x[i];                           /* update values */
}
```

It is important to have efficient sequential code. There are other ways the sequential algorithm could be coded that may be preferred for efficiency. We have used an `if` statement so as not to use `a[i][i] * x[j]` in the summation of `a[i][j] * x[j]`. Another solution to avoid the overhead of repeated `if` statements would be to include `a[i][j] * x[j]` in the loop and subtract it afterwards (or before):

```
for (i = 0; i < n; i++)
   x[i] = b[i];                                  /*initialize unknowns*/
for (iteration = 0; iteration < limit; iteration++) {
   for (i = 0; i < n; i++) {                     /* for each unknown */
      sum = -a[i][i] * x[i];
      for (j = 0; j < n; j++)                    /* compute summation */
         sum = sum + a[i][j] * x[j];
      new_x[i] = (b[i] - sum) / a[i][i];         /* compute unknown */
   }
   for (i = 0; i < n; i++) x[i] = new_x[i];      /* update values */
}
```

Yet another solution is to have two loops for computing the summation, the first from 0 to $i-1$ and the second from $i+1$ to $n-1$. We prefer our second version as a reasonably readable solution.

Parallel Code. Suppose that one process is allocated for each unknown ($p = n$) and each process will iterate the same number of times. On each iteration, the newly computed values of the unknowns need to be broadcast to all the other processes. In sequential code, the iteration `for` loop is a natural barrier between iterations. In parallel code, we need to insert a specific barrier. Process P_i could be of the form

```
x[i] = b[i];                                  /*initialize unknown*/
for (iteration = 0; iteration < limit; iteration++) {
   sum = -a[i][i] * x[i];
   for (j = 0; j < n; j++)                    /* compute summation */
      sum = sum + a[i][j] * x[j];
   new_x[i] = (b[i] - sum) / a[i][i];         /* compute unknown */
   broadcast_receive(&new_x[i]);              /* broadcast value */
   global_barrier();                          /* wait for all processes */
}
```

The broadcast routine, `broadcast_receive()`, sends the newly computed value of `x[i]` from process i to every other process and collects data broadcast from the other processes. A single broadcast will not work here since there must be matching "broadcast receives" in each process for each newly computed value. Hence, `broadcast_receive()` would have to consist of n broadcasts, each with specific parameters.

An alternative simple solution is to return to basic `send()`s and `recv()`s, for `broadcast_receive()`; that is, process i might have

```
for (j = 0; j < n; j++) if (i != j) send(&x[i], Pj);
for (j = 0; j < n; j++) if (i != j) recv(&x[j], Pj);
```

In MPI, it is allowable to send a message to yourself so that the `if` construct could be removed. A separate barrier may not be necessary since the process would not continue until it has received all the newly computed values.

Earlier, we saw how a butterfly barrier would naturally broadcast and gather values in one composite construction. The butterfly could be coded for our problem. However, since its efficiency depends upon the underlying architecture, a predefined routine would be helpful. MPI has such a routine, called `MPI_Allgather`. Allgather is illustrated in Figure 6.10. The same number of items are gathered from each process. The number is defined in the parameter list. A variation of `MPI_Allgather` called `MPI_Allgatherv` allows a different number of items to be gathered from each process.

Typically, we want to iterate until the approximations are sufficiently close, rather than for a fixed number of times (which may not provide a sufficiently accurate solution). Each process could check its own computed value with, say,

```
x[i] = b[i];                                /*initialize unknown*/
iteration = 0;
do {
   iteration++;
   sum = -a[i][i] * x[i];
   for (j = 0; j < n; j++)                  /* compute summation */
      sum = sum + a[i][j] * x[j];
   new_x[i] = (b[i] - sum) / a[i][i];       /* compute unknown */
   broadcast_receive(&new_x[i]);            /* broadcast value and wait */
} while (tolerance() && (iteration < limit));
```

where `tolerance()` returns `FALSE` if ready to terminate; otherwise it returns `TRUE`.

The simplest mechanism is to allow the processes to continue until they have all converged. Then `tolerance()` is a routine returning `FALSE` to each process in the same iteration, so that they all stop together. If we were to stop each process as it reaches its solution to the stated tolerance, the processes would have different numbers of iterations. In that case, some care would be needed here to avoid deadlock because broadcast routines broadcast to all the processes in a group and expect all the processes to have matching routines.

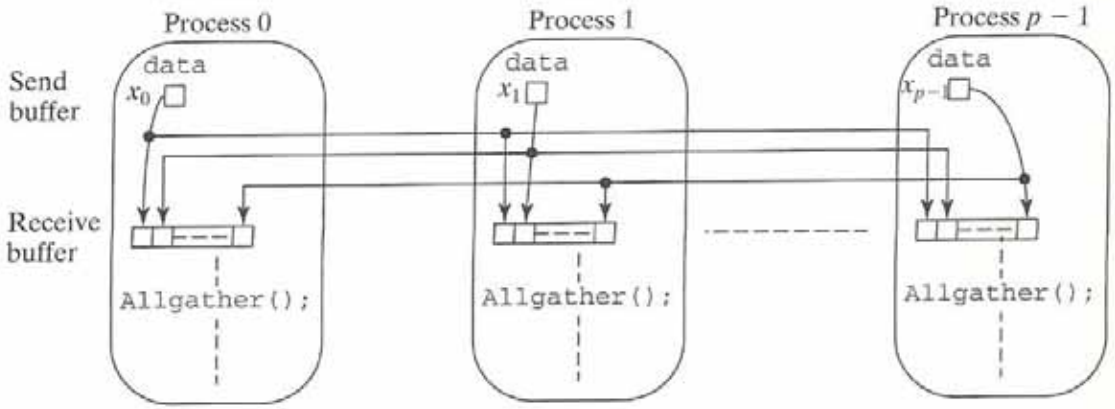


Figure 6.10 Allgather operation.

Partitioning. As with all parallel formulations, the number of processors is usually much smaller than the number of data items to be processed (computing unknowns in this case). We would normally partition the problem so that the processors take on more than one data item. In the problem at hand, each process can be responsible for computing a group of unknowns. Typically, we would allocate unknowns to processors in simple increasing order; that is, with p processors and n unknowns. Processor P_0 would be given the task of computing the unknowns x_0 to $x_{(n/p)-1}$, processor P_1 unknowns $x_{n/p}$ to $x_{(2n/p)-1}$, and so on, assuming that n/p is an integer — this is the so-called *block* allocation used in Chapter 4 for adding numbers. There is usually no advantage here in a *cyclic* allocation where processors are allocated one unknown in order; that is, processor P_0 is allocated $x_0, x_p, x_{2p}, \ldots, x_{((n/p)-1)p}$, processor P_1 is allocated $x_1, x_{p+1}, x_{2p+1}, \ldots, x_{((n/p)-1)p+1}$, and so on. Indeed, cyclic allocation may be disadvantageous because the indices of unknowns have to be computed in a more complex way, and it may require more effort to group the unknowns into one message.

Analysis. The sequential execution time of this problem will be given by the time of one iteration multiplied by the number of iterations. Suppose there are τ iterations. There are two loops, one nested inside the other. The outer loop has n iterations, and the inner loop has n^2 iterations in total. Each inner loop consists of one multiplication and one addition; that is, two computational steps. The outer loop has a multiplication and a subtraction prior to the inner loop, and a subtraction and division after the inner loop for a total of four computational steps. Thus, the sequential time is given by

$$t_s = n(2n + 4)\tau$$

which has a time complexity of $O(n^2)$ if there is a constant number of iterations.

The parallel execution time is the time of one processor when we assume that all processors execute the same number of iterations. Suppose there are n equations and p processors. A processor operates upon n/p unknowns. One iteration has a computational phase and a broadcast communication phase.

Computation. In the computational phase, there is an inner loop with n iterations and an outer loop with n/p iterations, both with the same computational effort as the nested sequential loops. Hence, the computation time is given by

$$t_{comp} = (n/p)(2n + 4)\tau$$

which has a time complexity of $O(n^2/p)$ if there is a constant number of iterations. As p is increased, the computation time decreases.

Communication. Communication occurs at the end of each iteration and consists of multiple broadcasts. In essence, all the n values computed by each processor must be relayed to every other processor. Given p separate broadcasts, each of the size n/p data items, and requiring t_{data} units of time to send each data item, the time could be of the form

$$t_{comm} = p(t_{startup} + (n/p)t_{data})\tau = (pt_{startup} + nt_{data})\tau$$

The communication time is a linearly increasing function of p, given a fixed value for n.

Overall Execution Time. The total parallel execution time is given by

$$t_p = ((n/p)(2n + 4) + pt_{startup} + nt_{data})\tau$$

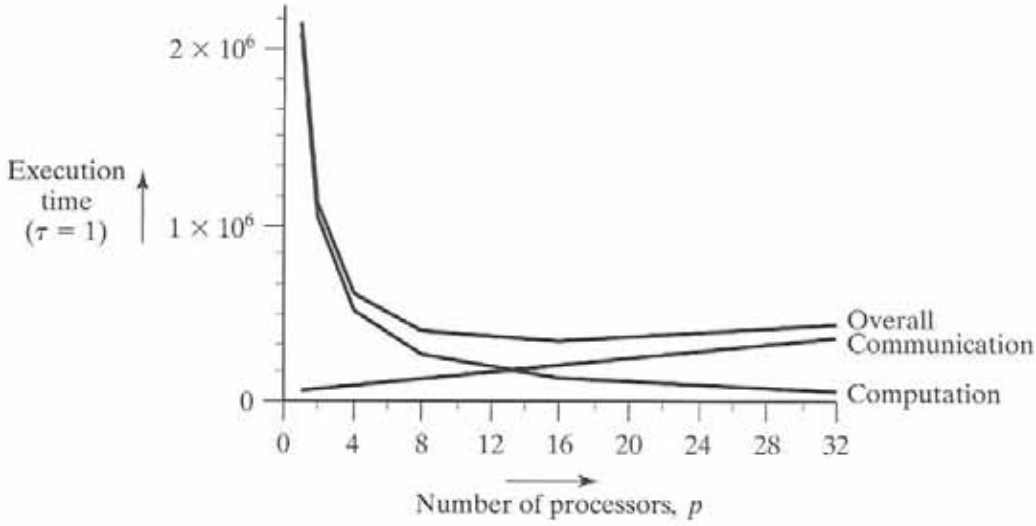


Figure 6.11 Effects of computation and communication in Jacobi iteration.

For non-negligible $t_{startup}$ times, the overall parallel execution time consists of one function that is a decreasing function of p (t_{comp}) and one function that is an increasing function of p (t_{comm}). The resulting total execution time has a minimum value. This characteristic is common to most partitioning situations. The minimum can be found by differentiation. To give a concrete example, suppose $t_{startup} = 10{,}000$ and $t_{data} = 50$ (representative of real systems; see Chapter 2). Figure 6.11 shows the overall execution time, the computation, and communication components, given that n/p must be an integer. The minimum execution time occurs when $p = 16$.

Speedup Factor. The speedup factor is given by

$$\text{Speedup factor} = \frac{t_s}{t_p} = \frac{n(2n+4)}{(n/p)(2n+4) + pt_{startup} + nt_{data}}$$

which is p if the communication is ignored. However, we have already established that there is an optimum number of processors for this problem, dependent upon the values for $t_{startup}$ and t_{data}.

Computation/communication Ratio. The computation/communication ratio is given by

$$\text{Computation/communication ratio} = \frac{t_{comp}}{t_{comm}} = \frac{(n/p)(2n+4)}{pt_{startup} + nt_{data}}$$

which suggests improvement with larger n (scalable).

6.3.2 Heat-Distribution Problem

The preceding problem required global synchronization. Now let us consider a local synchronization problem. Consider a square metal sheet that has known temperatures along each of its edges. The temperature of the interior surface of the sheet will depend upon the

temperatures around it. We can find the temperature distribution by dividing the area into a fine mesh of points, $h_{i,j}$. The temperature at an inside point can be taken to be the average of the temperatures of the four neighboring points, as illustrated in Figure 6.12. For this calculation, it is convenient to describe the edges by points adjacent to the interior points. The interior points of $h_{i,j}$ are where $0 < i < n$, $0 < j < n$ [$(n - 1) \times (n - 1)$ interior points]. The edge points are when $i = 0$, $i = n$, $j = 0$, or $j = n$, and have fixed values corresponding to the fixed temperatures of the edges. Hence, the full range of $h_{i,j}$ is $0 \le i \le n$, $0 \le j \le n$, and there are $(n + 1) \times (n + 1)$ points. We can compute the temperature of each interior point by iterating the equation

$$h_{i,j} = \frac{h_{i-1,j} + h_{i+1,j} + h_{i,j-1} + h_{i,j+1}}{4}$$

$(0 < i < n, 0 < j < n)$ for a fixed number of iterations or until the difference between iterations of a point is less than some very small prescribed amount.

This iteration equation occurs in several other similar problems; for example, with pressure and voltage. More complex versions appear for solving important problems in science and engineering. In fact, we are solving a system of linear equations. Each point is an unknown dependent upon a few other unknowns, rather than all the other unknowns in the general case. To clarify this relationship, consider the array of points as numbered in so-called *natural order* at the top left corner and in rows of m points, as shown in Figure 6.13. The points are numbered from 1 for convenience and include those representing the edges. (Note that $m = n$ here. However, m is used to differentiate the numbering system.) Each point will then use the equation

$$x_i = \frac{x_{i-1} + x_{i+1} + x_{i-m} + x_{i+m}}{4}$$

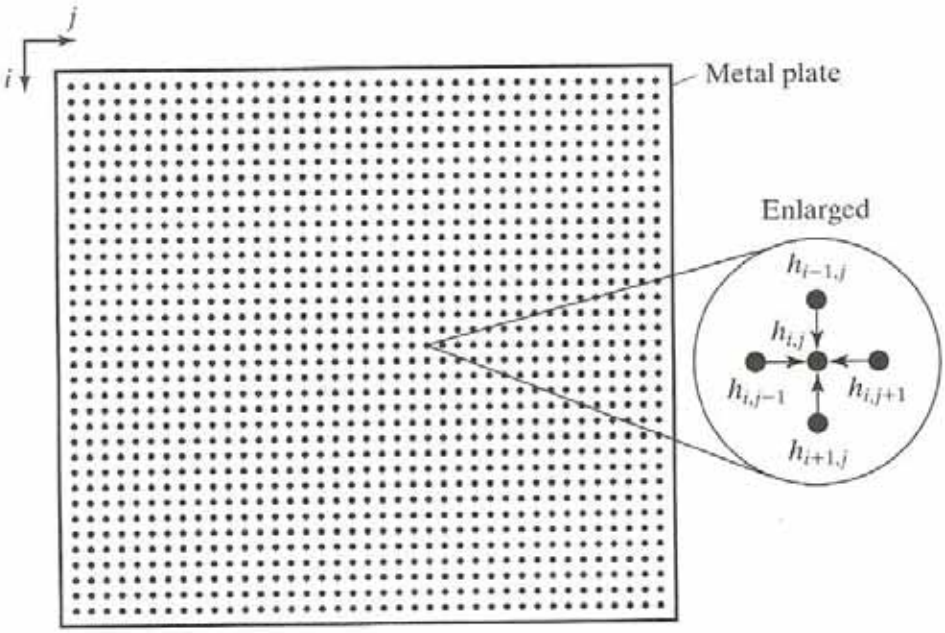


Figure 6.12 Heat distribution problem.

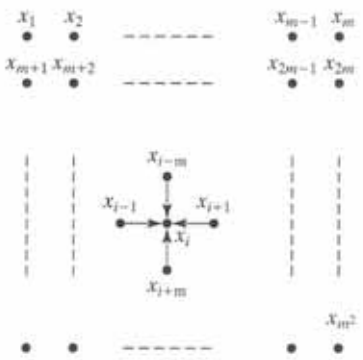


Figure 6.13 Natural ordering of heat-distribution problem.

This equation could be written as a linear equation containing the unknowns x_{i-m}, x_{i-1}, x_{i+1}, and x_{i+m}:

$$x_{i-m} + x_{i-1} - 4x_i + x_{i+1} + x_{i-m} = 0$$

$0 < i \leq m^2$, that is, m^2 equations. The method is known as the *finite difference* method. It can be extended into three dimensions by taking the average of six neighboring points, two in each dimension. We are also solving Laplace's equation. Chapter 11 explores further the relationship with linear equations. (In Chapter 11, n is used for the number of equations.)

Sequential Code. Returning to the original numbering system for the points, suppose that the temperature of each point is held in an array `h[i][j]`, and the boundary points `h[0][x]`, `h[x][0]`, `h[n][x]`, and `h[x][n]` ($0 \leq x \leq n$) have been initialized to the edge temperatures. The calculation as sequential code could be

```
for (iteration = 0; iteration < limit; iteration++) {
  for (i = 1; i < n; i++)
    for (j = 1; j < n; j++)
      g[i][j] = 0.25 * (h[i-1][j] + h[i+1][j] + h[i][j-1] + h[i][j+1]);
  for (i = 1; i < n; i++)              /* update points */
    for (j = 1; j < n; j++)
      h[i][j] = g[i][j];
}
```

using a fixed number of iterations. We multiply by 0.25 to compute the new value of the point rather than divide by 4 because multiplication is usually more efficient than division. Such normal methods to improve efficiency in sequential code carry over to parallel code and should be done where possible in all instances. (Of course, a good optimizing compiler would make such changes.)

There are several ways the code could be written if we want to stop at some precision, but in all cases, all points must have reached their precision. With properly initialized arrays, the sequential code could be

```
do {
  for (i = 1; i < n; i++)
    for (j = 1; j < n; j++)
      g[i][j] = 0.25*(h[i-1][j] + h[i+1][j] + h[i][j-1] + h[i][j+1]);
```

```
      for (i = 1; i < n; i++)          /* update points */
     for (j = 1; j < n; j++)
       h[i][j] = g[i][j];

    continue = FALSE;                  /* indicates whether to continue */
    for (i = 1; i < n; i++)            /* check each point for convergence */
      for (j = 1; j < n; j++)
        if (!converged(i,j) {        /* point found not converged */
          continue = TRUE;
          break;
        }

  } while (continue == TRUE);
```

Convergence is checked after all the points have been computed, which allows several possible convergence algorithms. The routine `converged(i,j)` returns `TRUE` if the element `g[i][j]` has converged to the required precision; otherwise it returns `FALSE`. The Boolean flag `continue` will be set to `TRUE` if at least one point in an iteration has not converged. Normally, we would want to ensure that the loop terminates even if convergence does not occur. This can also be done by incorporating a loop counter.

Improvements. In the above code, a second array, `g[][]`, is used to hold the newly computed values of the points from the old values. The array `h[][]` is updated with the new values held in `g[][]` after all the values have been computed in `g[][]`. Using the values of one iteration to compute the values for the next iteration in this fashion is known as a Jacobi iteration. A significant improvement is to eliminate the second array:

```
for (iteration = 0; iteration < limit; iteration++) {
  for (i = 1; i < n; i++)
    for (j = 1; j < n; j++)
      h[i][j] = 0.25 * (h[i-1][j] + h[i+1][j] + h[i][j-1] + h[i][j+1]);
```

Given the sequential order of the computation, two of the values used to compute `h[i][j]` have already been computed in that iteration (`h[i-1][j]` and `h[i][j-1]`) and are used, and two values have not yet been computed in that iteration and were computed from the preceding iteration (`h[i+1][j]` and `h[i][j+1]`). Thus, we are using the most recent values available. This is know as a Gauss-Seidel iteration and usually produces a faster convergence. However, it relies upon the sequential order of computation.

In the Gauss-Seidel method, the unknowns (points) to be computed, say x_i $(0 < i < n-1)$, are ordered so that those before the current unknown, x_j, $j < i$, have already been computed for that iteration and are used, and those after the current point, x_k, $k > i$, have not yet been computed in the current iteration, and therefore the values computed in the preceding iteration are used. The basic Gauss-Seidel method is an excellent match for a sequential program in which unknowns are computed in some sequential order, but as described it is not a good basis for parallel program in which unknowns are computed simultaneously. However, there are specific orderings that allow for simultaneous computations. We shall explore different orderings and faster iteration methods that are

amenable to parallelization at the end of this chapter and in more detail in Chapter 11. For now, we shall concentrate upon the Jacobi iteration because it allows us to investigate local synchronization, but one should be aware that it is usually neither the best sequential algorithm nor the best basis for a parallel algorithm.

Parallel Code. The sequential code is "unnatural" in that we have used `for` loops to visit each point, whereas the points can be visited simultaneously without any change to the algorithm:

```
for (iteration = 0; iteration < limit; iteration++) {
  forall (i = 1; i < n; i++)
    forall (j = 1; j < n; j++)
      h[i][j] = 0.25 * (h[i-1][j] + h[i+1][j] + h[i][j-1] + h[i][j+1]);
}
```

In this construction, it is understood that all the values on the right side of the computation are computed from the preceding iteration without any need of an explicit array `g[][]`. We would usually partition the problem so that more than one point is processed by each process. However, in the first instance, suppose that we have one process for each point.

Each process requires the four neighboring points, and the most convenient organization is to arrange the processes conceptually into a mesh. Let us refer to processes by subscripts on the mesh by using row major order, where the first subscript is the row, and the second, the column. For the version with a fixed number of iterations, process $P_{i,j}$ (except for the boundary points) could be of the form

```
for (iteration = 0; iteration < limit; iteration++) {
   g = 0.25 * (w + x + y + z);
   send(&g, P_{i-1,j});              /* non-blocking sends */
   send(&g, P_{i+1,j});
   send(&g, P_{i,j-1});
   send(&g, P_{i,j+1});
   recv(&w, P_{i-1,j});              /* synchronous receives */
   recv(&x, P_{i+1,j});
   recv(&y, P_{i,j-1});
   recv(&z, P_{i,j+1});
}
```

Local barrier

after suitable initialization of `w`, `x`, `y`, and `z`. Each process has its own iteration loop. The number of iterations must be sent to each process. It is important to use `send()`s that do not block while waiting for the `recv()`s; otherwise the processes would deadlock, each waiting for a `recv()` before moving on. The `recv()`s must be synchronous and wait for the `send()`s. Each process will be synchronized with its four neighbors by the `recv()`s. We are using a local synchronization technique here. It is unnecessary to have a separate iteration to update the array. The transfer between four processes is shown in Figure 6.14.

Implementing the version where processes stop when they reach their required precision requires a master process to be notified when all the (slave) processes have

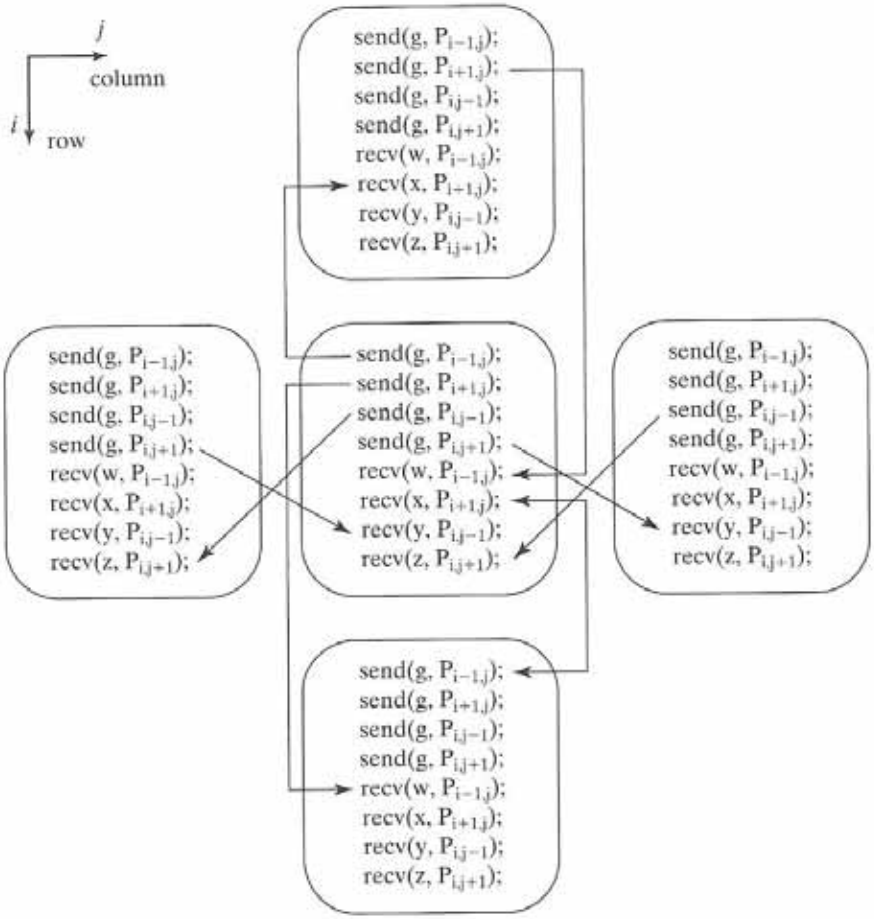


Figure 6.14 Message-passing for heat-distribution problem.

stopped. A process could send data to the master when the precision has been reached locally; for example,

```
iteration = 0;
do {
   iteration++;
   g = 0.25 * (w + x + y + z);
   send(&g, P_{i-1,j});                    /* locally blocking sends */
   send(&g, P_{i+1,j});
   send(&g, P_{i,j-1});
   send(&g, P_{i,j+1});
   recv(&w, P_{i-1,j});                    /* locally blocking receives */
   recv(&x, P_{i+1,j});
   recv(&y, P_{i,j-1});
   recv(&z, P_{i,j+1});
} while((!converged(i, j)) && (iteration < limit));
send(&g, &i, &j, &iteration, P_master);
```

To handle the processes operating at the edges, we could use the process ID to determine the location of the process in the array, leading to code such as

```
if (last_row) w = bottom_value;
if (first_row) x = top_value;
if (first_column) y = left_value;
if (last_column) z = right_value;
iteration = 0;
do {
   iteration++;
   g = 0.25 * (w + x + y + z);
   if !(first_row) send(&g, P_{i-1,j});
   if !(last_row) send(&g, P_{i+1,j});
   if !(first_column) send(&g, P_{i,j-1});
   if !(last_column) send(&g, P_{i,j+1});
   if !(last_row) recv(&w, P_{i-1,j});
   if !(first_row) recv(&x, P_{i+1,j});
   if !(first_column) recv(&y, P_{i,j-1});
   if !(last_column) recv(&z, P_{i,j+1});
} while((!converged) && (iteration < limit));
send(&g, &i, &j, iteration, P_master);
```

It is a simple matter for us to convert the process indices to actual numbers. Given processes numbered starting at the top left corner of the mesh and numbered across rows (natural ordering), process P_i communicates with P_{i-1} (left), P_{i+1} (right), P_{i-k} (up), and P_{i-k} (down) ($0 \le i < k^2$).

Partitioning. Obviously, we would normally allocate more than one point to each processor, because there would be many more points than processors. The mesh of points could be partitioned into square blocks or strips (columns), as shown in Figure 6.15 (p partitions). The partitions are the same options as for the image bitmap in Chapter 3, but now there is communication between partitions. Therefore, we would like to minimize the communication. With n^2 points, p processors, and equal partitions, each partition holds n^2/p points. The communication consequences of the two arrangements are shown in Figure 6.16.

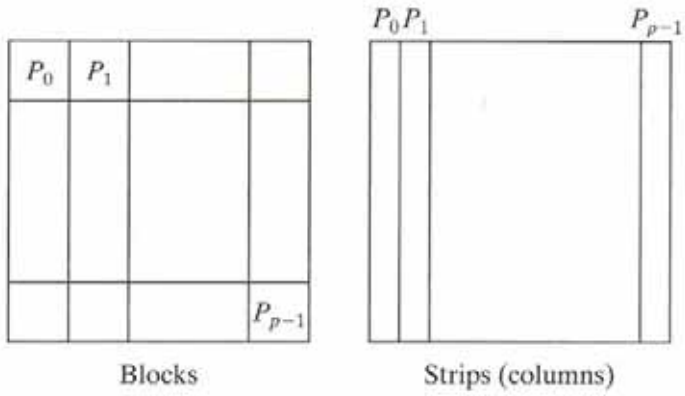


Figure 6.15 Partitioning heat-distribution problem.

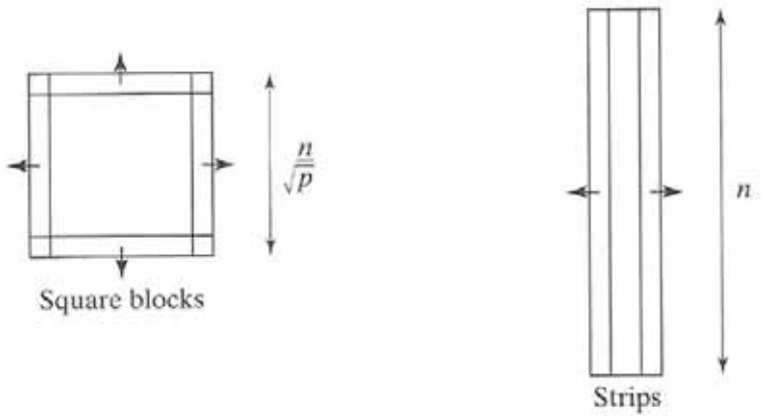


Figure 6.16 Communication consequences of partitioning.

In the block partition, there are four edges where data points are exchanged. Each process generates four messages and receives four messages in each iteration (assuming that the data points along one edge are packed into one message). Thus, the communication time is given by

$$t_{\text{commsq}} = 8\left(t_{\text{startup}} + \frac{n}{\sqrt{p}} t_{\text{data}}\right)$$

This equation is only valid for $p \geq 9$ when at least one block has four communicating edges.

In the strip partition, there are two edges where data points are exchanged. Each process generates two messages and receives two messages in each iteration (assuming again that all the data points along one edge are packed into one message). Thus, the communication time is given by

$$t_{\text{commcol}} = 4(t_{\text{startup}} + n t_{\text{data}})$$

Note that the communication time is independent of the number of processors p; that is, the number of partitions.

These communication times will be heavily influenced by the startup time. For example, suppose $t_{\text{startup}} = 10{,}000$, $t_{\text{data}} = 50$ (as used for Figure 6.11), and $n^2 = 1024$. Under these circumstances, the strip partition has a communication time of 46,400 (time units) irrespective of the value of p. The block partition has a communication time of $80{,}000 + 12800/\sqrt{p}$. For any number of processors, this will be greater than the strip partition. But suppose the startup time is 100. The strip partition now has a communication time of 6800, whereas the block partition has a communication time of $800 + 12800/\sqrt{p}$. Now the strip partition always has a greater communication time for $p > 4$.

In general, the strip partition is best for a large startup time, and a block partition is best for a small startup time. With the previous equations, the block partition has a larger communication time than the strip partition if

$$8\left(t_{\text{startup}} + \frac{n}{\sqrt{p}} t_{\text{data}}\right) > 4(t_{\text{startup}} + n t_{\text{data}})$$

or

$$t_{\text{startup}} > n\left(1 - \frac{2}{\sqrt{p}}\right)t_{\text{data}}$$

($p \geq 9$). The right side tends to nt_{data}, or 1600 with our numbers, as p increases modestly. For example, the crossover point between block partition being best and strip partition being best is reached with $p = 64$ when $t_{\text{startup}} = 1200$. Figure 6.17 shows the characteristics for $p = 9$, 16, 64, 256, and 1024.

The startup time will be large in most systems, especially in workstation clusters. Hence, the strip partition seems the appropriate choice. It is also the easiest partition for us to program, since messages simply pass left and right rather than left, right, up, and down.

Implementation Details. It will be necessary for us to send a complete column of points to an adjacent process in one message. For convenience, we might divide the two-dimensional array of points into rows rather than columns when the array is stored in row major order (as in C). Then a row can be sent in a message simply by specifying the starting address of the row and the number of data elements stored in it (a contiguous group of elements). If we do not want to utilize the actual storage structure, then a separate one-dimensional array could be used to hold the points being transmitted to an adjacent process. We could have started our discussion by dividing into rows rather than columns, but this implementation detail is not an algorithmic matter.

In addition to the division of points into rows, Figure 6.18 shows each process having an additional row of points at each edge, called *ghost points*, that hold the values from the adjacent edge. Each array of points is increased to accommodate the ghost rows. Ghost points are provided for programming convenience.

The code for process P_i (not including the processes at the borders) could take the form

```
for (k = 1; k <= n/p; k++)        /* compute points in partition */
  for (j = 1; j <= n; j++)
   g[k][j] = 0.25 * (h[k-1][j] + h[k+1][j] + h[k][j-1] + h[k][j+1]);
```

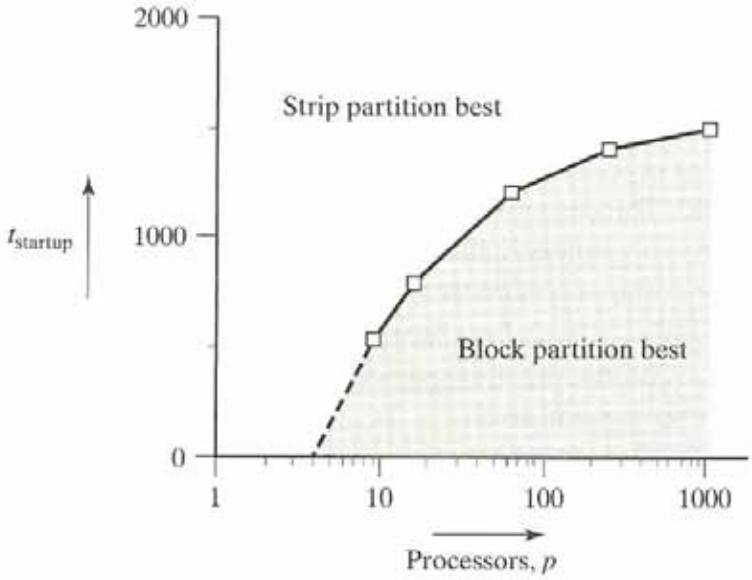


Figure 6.17 Startup times for block and strip partitions.

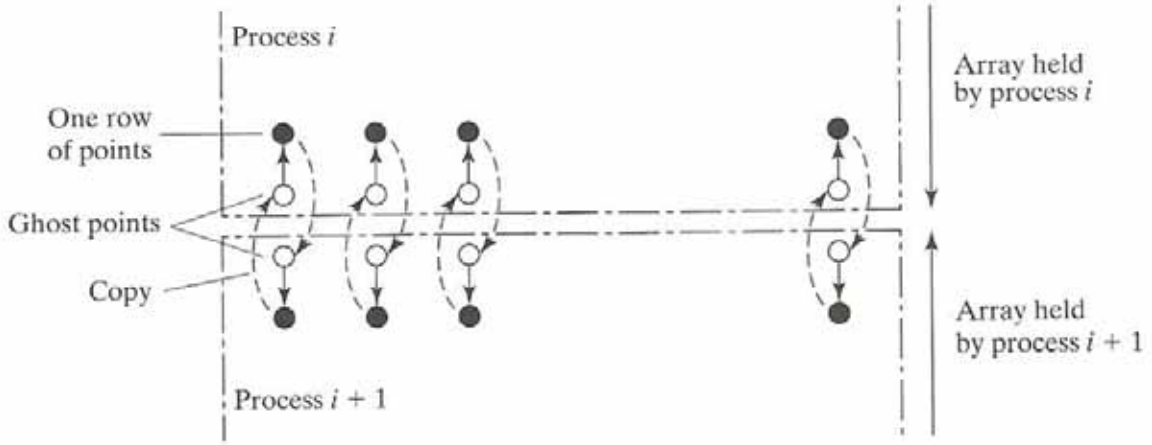


Figure 6.18 Configurating array into contiguous rows for each process, with ghost points.

```
for (k = 1; k <= n/p; k++)          /* update points */
  for (j = 1; j <= n; j++)
   h[k][j] = g[k][j];
send(&g[1][1], n, P_{i-1});         /* send row to adjacent process */
send(&g[n/p][1], n, P_{i+1});
recv(&h[0][1], n, P_{i-1});         /* receive row from adjacent process */
recv(&h[n/p + 1][1], n, P_{i+1});
```

Safety and Deadlock. The arrangement when all the processes send their messages first and then receive all of their messages, as in all the code so far, is described as unsafe in the MPI literature. This is because it relies upon buffering in the `send()`s. The amount of buffering is not specified in MPI. If a send routine has insufficient storage available when it is called, the implementation should be such as to delay the routine from returning until storage becomes available or until the message can be sent without buffering. Hence, the locally blocking `send()` could behave as a synchronous `send()`, only returning when the matching `recv()` is executed. Since a matching `recv()` would never be executed if all the `send()`s are synchronous, deadlock would occur. In our case, it is likely that sufficient storage is available if n/p is relatively small — and you might question why an implementation would have a locally blocking send available if it cannot be used with safety!

A way of making the code safe is to alternate the order of the `send()`s and `recv()`s in adjacent processes. This is so that only one process performs the `send()`s first. Then even synchronous `send()`s would not cause deadlock. In fact, a good way you can test for safety is to replace message-passing routines in a program with synchronous versions.

Safe code, by alternating the `send()`s and `recv()`s, could be of the form

```
if ((i % 2) == 0) {                 /* even-numbered processes */
   send(&g[1][1], n, P_{i-1});
   recv(&h[0][1], n, P_{i-1});
   send(&g[n/p][1], n, P_{i+1});
   recv(&h[n/p + 1][1], n, P_{i+1});
} else {                            /* odd-numbered processes */
   recv(&h[n/p + 1][1], n, P_{i+1});
   send(&g[n/p][1], n, P_{i+1});
```

```
    recv(&h[0][1], n, P_{i-1});
    send(&g[1][1], n, P_{i-1});
}
```

Alternating the `send()`s and `recv()`s can easily be done here but could be more difficult in other circumstances.

MPI offers several alternative methods for safe communication:

- Combined send and receive routines: `MPI_Sendrecv()` (which is guaranteed not to deadlock)
- Buffered send()s: `MPI_Bsend()` — here the user provides explicit storage space
- Nonblocking routines: `MPI_Isend()` and `MPI_Irecv()` — here the routine returns immediately, and a separate routine is used to establish whether the message has been received (`MPI_Wait()`, `MPI_Waitall()`, `MPI_Waitany()`, `MPI_Test()`, `MPI_Testall()`, or `MPI_Testany()`)

A pseudocode segment using the third method is

```
isend(&g[1][1], n, P_{i-1});
isend(&g[n/p][1], n, P_{i+1});
irecv(&h[0][1], n, P_{i-1});
irecv(&h[n/p + 1][1], n, P_{i+1});
waitall(4);
```

Essentially, the wait routine becomes a barrier, waiting for all the message-passing routines to complete.

6.3.3 Cellular Automata

A concept that is particularly suitable for synchronous iteration is called *cellular automaton*. In cellular automation, the problem space is first divided into cells. Each cell can be in one of a finite number of states. Cells are affected by their neighbors according to certain rules, and all cells are affected simultaneously in a "generation." The rules are reapplied in subsequent generations so that cells evolve, or change state, from generation to generation.

The most famous cellular automaton is the "Game of Life," devised by John Horton Conway, a Cambridge mathematician, and published by Gardner (Gardner, 1967). Gardner points out that the concept of cellular automata can be traced back to Von Neuman's work in the early 1950s. The Game of Life is a board game; the board consists of a (theoretically infinite) two-dimensional array of cells. Each cell can hold one "organism" and has eight neighboring cells, including those diagonally adjacent. Initially, some of the cells are occupied in a pattern. The following rules apply:

1. Every organism with two or three neighboring organisms survives for the next generation.
2. Every organism with four or more neighbors dies from overpopulation.

3. Every organism with one neighbor or none dies from isolation.
4. Each empty cell adjacent to exactly three occupied neighbors will give birth to an organism.

These rules were derived by Conway "after a long period of experimentation."

Another simple fun example of cellular automata is "Sharks and Fishes" in the sea, each with different behavioral rules. A two-dimensional version of this problem is studied in detail in Fox et al. (1988). An ocean could be modeled as a three-dimensional array of cells. Each cell can hold one fish or one shark (but not both). Fish might move around according to these rules:

1. If there is one empty adjacent cell, the fish moves to this cell.
2. If there is more than one empty adjacent cell, the fish moves to one cell chosen at random.
3. If there are no empty adjacent cells, the fish stays where it is.
4. If the fish moves and has reached its breeding age, it gives birth to a baby fish, which is left in the vacated cell.
5. Fish die after x generations.

The sharks might be governed by the following rules:

1. If one adjacent cell is occupied by a fish, the shark moves to this cell and eats the fish.
2. If more than one adjacent cell is occupied by a fish, the shark chooses one fish at random, moves to the cell occupied by the fish, and eats the fish.
3. If there are no fish in adjacent cells, the shark chooses an unoccupied adjacent cell to move to in the same way that fish move.
4. If the shark moves and has reached its breeding age, it gives birth to a baby shark, which is left in the vacated cell.
5. If a shark has not eaten for y generations, it dies.

Problem 6-21 describes a similar problem with foxes and rabbits. The behavior of the rabbits is to move around happily, whereas the behavior of the foxes is to eat any rabbits they come across.

There are serious applications for cellular automata, because they avoid the need for differential equations. For example, given the rules of fluid/gas dynamics, the movement of fluids and gases around objects or diffusion of gases can be modeled by this method. Biological growth can also be modeled. Examples given in the problems include airflow across an airplane wing (Problem 6-24) and erosion/movement of sand at a beach or riverbank (Problem 6-23). No doubt there are many other possible applications for cellular automata (Problem 6-22).

6.4 PARTIALLY SYNCHRONOUS METHODS

It is clear that synchronization causes a significant degradation of performance. In this final section, we will explore ways to reduce the amount of synchronization in the synchronous iteration problems explored earlier. Let us take the heat-distribution problem as an example.

The parallel code was written as

```
for (iteration = 0; iteration < limit; iteration++) {
  forall (i = 1; i < n; i++)
    forall (j = 1; j < n; j++)
      h[i][j] = 0.25 * (h[i-1][j] + h[i+1][j] + h[i][j-1] + h[i][j+1]);
}
```

with the assumption that the values on the right side of the computation are computed from the preceding iteration. The code computes the next iteration values based on the immediately preceding iteration values. This is the traditional Jacobi iteration method, which requires a global synchronization point (barrier) for processes to wait until all processes have performed their calculations. It is not the calculation that causes the performance reduction but the time it takes to perform the barrier synchronization. We have already mentioned the possibility of using some of the present iteration values in the calculation, as in the Gauss-Seidel iteration method, but a barrier is still present because all the processes operate on the same iteration together. Suppose the barrier was removed altogether, allowing processes to continue with subsequent iterations before other processes have completed their present iteration. Then the processes moving forward would use values computed from not only the preceding iteration but from earlier iterations and not only the last iteration. The method is called an *asynchronous iterative method*. The mathematical conditions for convergence may be more strict in asynchronous iteration; that is, the calculation may not converge unless certain mathematical conditions exist. Each process may not be allowed to use any previous iteration values if the method is to converge. A form of asynchronous iterative method called *chaotic relaxation* was introduced by Chazan and Miranker (1969) in which the convergence conditions are stated as:

> "there must be a fixed positive integer s such that, in carrying out the evaluation of the ith iterate, a process cannot make use of any value of the components of the jth iterate if $j < i - s$" (Baudet, 1978).

This suggests a simple alteration to the parallel code to allow processes to continue until they try to use a value which is more than s iteration cycles previously. Each stored value will need a "time stamp," the iteration number associated with the stored value.

The final part of the code, checking for convergence of every iteration, can also be reduced. It may be better to allow iterations to continue for several iterations before checking for convergence. Combining the chaotic relaxation and delayed convergence testing, each process is allowed to perform s iterations before being synchronized but will also update its locally stored values as it goes. At every s iterations, the maximum divergence is recorded. Convergence is checked then. The actual iteration corresponding to the elements of the array being used at any time may be from an earlier iteration but only up to s iterations previously. In a message-passing solution, all the data values that are obtained from other processes will be from s iterations previously when the processes last communicated. Data values being used that are computed within the process will be from the present iteration or the previous iteration, depending upon the sequence in which the data values are computed sequentially.

Note that the method described cannot be applied to all synchronous problems. For example, cellular automation does not lend itself to this approach.

6.5 SUMMARY

This chapter introduced the following:

- The concept of a barrier and its implementations (global barriers and local barriers)
- Data parallel computations
- The concept of synchronous iteration
- Examples of using global and local barriers
- The concept of safe communication
- Cellular automata
- Partially synchronous methods

FURTHER READING

The concept of a barrier is discussed in most parallel programming texts. Apart from the software implementations of barriers described here, some multiprocessor systems (e.g., the CRAY T3D) have hardware support for barriers. Pacheco (1997) develops MPI code for Jacobi iterations. MPI code for Jacobi iterations can also be found in Snir et al. (1996), with significant discussion about safe programs and alternative coding when you use special features of MPI, such as the `MPI_Sendrecv()` routine, posting routines, and null processes. Significant MPI details for Jacobi iterations can also be found in the other MPI "reference," Gropp, Lusk, and Skjellum (1999), which also includes the use of MPI features such as topologies. A discussion of the trade-off between computation and communication can be found in Snir et al. (1996) and Wilson (1995). In addition to the fully synchronous technique discussed in this chapter, synchronization can be applied in a less structured or loosely synchronous manner, whereby processes are synchronized occasionally. Several loosely synchronous applications are discussed in detail in Fox, Williams, and Messina (1994).

Since its introduction by Chazan and Miranker (1969), chaotic relaxation has been studied by several authors, including Baudet (1978) and Evans and Yousif (1992), although it was ignored in textbooks on parallel programming and algorithms in the past. It offers the potential for very significant improvement in execution speed over fully synchronous methods, when it can be applied.

BIBLIOGRAPHY

BAUDET, G. M. (1978), Asynchronous Iterative Methods for Multiprocessors, *J. ACM*, Vol. 25, pp 226–244.

BERTSEKAS, D. P., AND J. N. TSITSIKLIS (1989), *Parallel and Distributed Computation Numerical Methods*. Prentice Hall, Englewood Cliffs, NJ.

CHAZAN, D, AND W. MIRANKER (1969), Chaotic Relaxation, *Linear Algebra and Its Applications*, Vol 2, pp. 199–222.

EVANS, D. J, AND N. Y. YOUSIF (1992), "Asynchronous Parallel Algorithms for Linear Equations," in *Parallel Processing in Computational Mechanics* (editor H. Adeli), Marcel Dekker, NY, pp. 69–130.

FOX, G. C., M. A. JOHNSON, G. A. LYZENGA, S. W. OTTO, J. K. SALMON, AND D. W. WALKER (1988), *Solving Problems on Concurrent Processors*, Volume 1, Prentice Hall, Englewood Cliffs, NJ.

FOX, G. C., R. D. WILLIAMS, AND P. C. MESSINA (1994), *Parallel Computing Works*, Morgan Kaufmann, San Francisco, CA.

GARDNER, M. (1967), "Mathematical Games," *Scientific American*, October, pp. 120–123.

GROPP, W., E. LUSK, AND A. SKJELLUM (1999), *Using MPI Portable Parallel Programming with the Message-Passing Interface*, MIT Press, Cambridge, MA.

HILLIS, W. D., AND G. L. STEELE, JR. (1986), "Data Parallel Algorithms," *Comm. ACM*, Vol. 29, No. 12, pp. 1170–1183.

JÁJÁ, J. (1992), *An Introduction to Parallel Algorithms*, Addison Wesley, Reading, MA.

PACHECO, P. (1997), *Parallel Programming with MPI*, Morgan Kaufmann, San Francisco, CA.

SNIR, M., S. W. OTTO, S. HUSS-LEDERMAN, D. W. WALKER, AND J. DONGARRA (1996), *MPI: The Complete Reference*, MIT Press, Cambridge, MA.

WAGNER, R. A., AND Y. HAN (1986), "Parallel Algorithms for Bucket Sorting and the Data Dependent Prefix Problem," *Proc. 1986 Int. Conf. Par. Proc.*, IEEE CS Press, pp. 924–929.

WILSON, G. V. (1995), *Practical Parallel Programming*, MIT Press, Cambridge, MA.

PROBLEMS

Scientific/Numerical

6-1. Implement the counter barrier described in Figure 6.4, and test it. Is it necessary to use blocking or synchronous routines for both send and receive? Explain.

6-2. Write a barrier, `barrier(procNum)`, which will block until `procNum` processes reach the barrier and then release the processes. Allow for the barrier to be called with different numbers of processes and with different values for `procNum`.

6-3. Investigate the time that a barrier takes to operate by using code such as

```
t1 = time();
Barrier(group);
t2 = time();
printf("Elapsed time = %", difftime(t2, t1));
```

(In MPI the barrier routine is `MPI_Barrier(Communicator)`. The time routine is `MPI_Wtime()`.) Investigate different numbers of processes.

6-4. Write code to implement an eight-process barrier using the tree construction described in Section 6.1.3 and compare with any available barrier calls (e.g., in MPI `MPI_Barrier()`).

6-5. Implement the butterfly barrier described in Section 6.1.4, and compare with any available barrier calls.

6-6. Determine experimentally at what point in your system the limit to buffering is reached when using nonblocking sends. Establish the effects of requesting more buffering than is available. (It may be that the amount of buffering available is related to the amount of memory being used for other purposes.)

6-7. Can noncommutative operators such as division be used in the prefix calculation of Figure 6.8?

6-8. Determine the efficiency of the prefix calculation of Figure 6.8.

6-9. Given a fixed rectangular area with sides x and y and a communication that is proportional to the perimeter, $2(x + y)$, show that the minimum communication is given by $x = y$ (i.e., a square).

6-10. Write a parallel program to solve the one-dimensional problem based upon the finite difference equation

$$x_i = \frac{x_{i-1} + x_{i+1}}{2}$$

for $0 \le i \le 1000$, given that $x_0 = 10$ and $x_{1000} = 250$.

6-11. In the text, we have assumed a square array for the heat-distribution problem of Section 6.3.2. What are the mathematical conditions for choosing blocks or strips as the partition if the array has a length of n points and a width of m points?

6-12. Investigate the accuracy of convergence of the heat-distribution problem using the different termination methods described in Section 6.3.1. Determine whether it is sufficient to use the difference between the present and next values of the points or whether it is necessary to use a more complex termination calculation. The basic question being investigated here is, "If each point is computed until each is within 1 percent (say) of its previous computed value, what is the accuracy of the solution?"

6-13. Write a parallel program to simulate the Game of Life as described in Section 6.3.3 and experiment with different initial populations.

6-14. Compare experimentally a fully synchronous implementation and a partially synchronous implementation of the heat-distribution problem described in Section 6.3.1. Try different values of s in the convergence condition specified in Section 6.4. Write a report on your findings that includes the specific speed improvements you obtained.

Real Life

6-15. Figure 6.19 shows a room that has four walls and a fireplace. The temperature of the wall is 20°C, and the temperature of the fireplace is 100°C. Write a parallel program using Jacobi iteration to compute the temperature inside the room and plot (preferably in color) the temperature contours at 10°C intervals using Xlib calls or similar graphics calls as available on your system. Instrument the code so that the elapsed time is displayed. (This programming assignment is convenient after a Mandelbrot assignment because it can use the same graphics calls.)

6-16. Repeat Problem 6-15 but with a round room of diameter 20 ft and a point heat source in the center at 100°C; the walls are at 20°C.

6-17. Simulate a road junction controlled by traffic lights, as shown in Figure 6.20. Vehicles come from all four directions along the roads and wish either to pass straight through the junction to the other side, or turn left, or turn right. On average, 70 percent of the vehicles wish to pass straight through, 10 percent wish to turn right, and 20 percent wish to turn left. Each vehicle moves at the same speed up to the junction. Develop a set of driving rules to solve this problem by a cellular automata approach, and implement them in a parallel program using your own test data (vehicle numbers and positions).

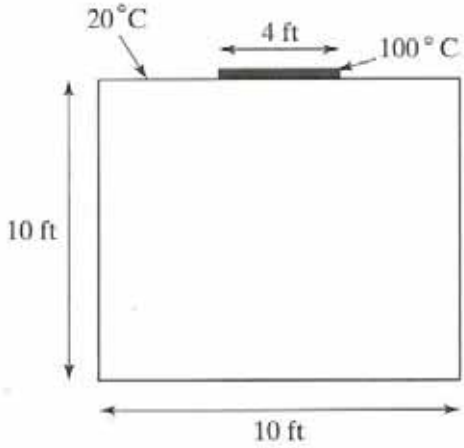


Figure 6.19 Room for Problem 6-15.

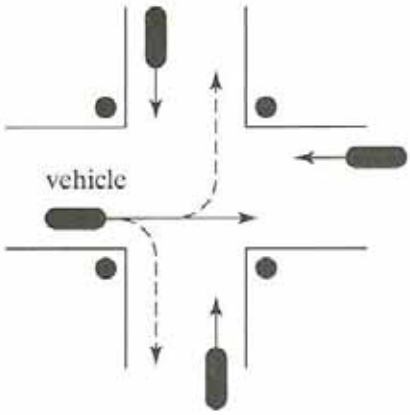


Figure 6.20 Road junction for Problem 6-17.

6-18. Write a parallel program to simulate the actions of the sharks and fish as described in Section 6.3.3. The parameters that are input are size of ocean, number of fish and sharks, their initial placement in the ocean, breeding ages, and shark starvation time. Adjacent cells do not include diagonally adjacent cells. Therefore, there are six adjacent cells, except for the edges. For every generation, the fishes' and sharks' ages are incremented by 1. Modify the simulation to take into account currents in the water.

6-19. Dr. Michaels was known across campus for being somewhat absent-minded. Thus, it was no surprise when he went on a camping trip into the Uwharrie National Forest, but left his map and compass behind. Luckily for him, he had packed his new portable computer that included one of the new cellular modems. Luckier still, he left you sitting back in the computer science building working on a research project!

You had a premonition about this trip, so you downloaded a detailed map of the forest that showed the location of every tree, cliff, and road/path through the forest from the latest NASA satellite pictures. The data is in the form of an array of "cells," with rectangular areas of 0.3 meters on an edge. Each cell contains a 'T', 'C', 'O', or 'R,' which designates what is in that area of the forest:

'T' The area contains an impassable tree.

'C' The area contains an abrupt drop-off (a cliff).

'O' The area is open and passable by the professor.

'R' The area is a roadway or marked path and is passable by the professor.

Thus, you were not particularly surprised when Dr. Michaels's e-mail message came through asking for your help. It seems he is suffering from a medical condition that necessitates his exiting the forest as rapidly as possible. He has asked you to write a program to do two things:

1. Identify where in the forest he happens to be.
2. Direct him out of the forest anywhere along the road that borders the southern edge.

Your program can query the professor about what is immediately ahead, behind, left, and right of his present cell. In response to each query, he will send back four letters. For example, the results of Query() might be 'T', 'O', 'O', 'C' and indicate that he is facing a tree blocking his movement forward, that he can move backward or to his left, and that he finds an impassable cliff to his right. By implication, the cell he is standing in contains 'O' or 'R.'

Your program can tell the professor to move one cell in any of the four directions by sending him an 'F' (move forward), a 'B' (move backward), 'L' (move left), or 'R' (move right). The syntax is Move('L') for a move left, and so on. Keep in mind that if you tell him to move into a cell containing a tree, your grade will suffer when he finally returns. However, if you direct him to move into a cell containing a cliff, it will not only be on your conscience but also appear as an 'F' on your grade report (which will be filed by Dr. Michaels's next of kin).

Your program is to be able to identify the professor's location in as few query/movement combinations as possible, and then direct him by way of the shortest route possible from that location to the road that runs along the forest's southern boundary.
Prototypes of the Query() and Move() functions are as follows:

```
char * Query(void)
/* Query returns a pointer to a string of four characters */

int Move(char direction);
/* if the move is successful, Move returns the value 0. If it is unsuccessful
because you directed him into a tree, Move returns a -1. If Move is
unsuccessful because you directed the professor off a cliff, Move returns a
-100 indicating you just failed your research project work and need to call
the coroner. */
```

Hint: The professor may be facing any of four directions, north, east, south, or west, and does not have a compass. Thus, you will have to match the pattern he returns in response to a Query() to your map data in four possible orientations to narrow down the set of possible locations he is in. Then you will have to Move() him and again Query() him. When you have finally determined where he is located, you have to find the shortest route out of the forest to the southern boundary road.

6-20. Eric was fascinated by the latest episode of "Who Done It?", a mystery thriller he had watched on tape-delay last night. It seems the key to solving the mystery was the ability of Sam Shovel, the detective, to match patterns in various handwriting samples. Eric decided to write a simple program to mimic Sam's pattern-matching behavior. The first thing Eric did was to create a set of 26 "perfect" printed letters on 15×21 grids. These templates would then be compared to actual printing samples, one after the other, to deduce the actual printed characters. His first attempt at writing this program was a total flop! He soon had discovered that none of the actual printed characters was an exact match for his "perfect" characters; he had not recognized any part of the suspect's message.

He then decided to try three radically different approaches. In the first, he used a pipelined solution: scaled the character to a nominal size, centered it in a grid, determined its axes of symmetry, and rotated it to a standard orientation; then compared it to the set of "perfect" characters. In the second, he smoothed the printed character to eliminate noise from

the suspect's jittery printing by blurring it slightly using a mathematics-based filter operation, by applying still more mathematics to look at the character in a transform domain, and finally by comparing that to the transforms of the "perfect" characters. In the third approach, Eric decided to simplify things still further; he just counted the number of matches between cells on the 15×21 grid and the grid containing the printed character. He moved the printed character around over the "perfect" one until he got the best match, recorded the number of matches, and then repeated with the next "perfect" character until all 26 had been compared; the best match must be the winner, he thought.

Give a brief analysis of each of his approaches with respect to the one with the best prospects for parallel processing.

6-21. Once upon a time there was an island populated only by rabbits, foxes, and vegetation. The island (conveniently enough!) was the exact shape of a chessboard. Some local geographers have even drawn gridlines that serve to divide the island into 64 squares to facilitate their demographic studies on the populations of each inhabitant.

Within each square the populations of rabbits and foxes are governed by several factors:

- the populations of rabbits and foxes in each square at the start of this "day"
- the reproduction rates of rabbits and foxes (the same over the entire island) during this day
- the vegetation growth rate
- the death rates of "old" rabbits and foxes during this day
- the eating habits of foxes (foxes live entirely on rabbits; when the vegetation is dense, rabbits are more difficult to find)
- the eating habits of rabbits (they live on the vegetation; too many rabbits in a square could lead to their starving and/or a lower reproduction rate and/or being easier for foxes to find and eat)
- the migration (from day to day) of rabbits from one square to other squares that are immediately adjacent
- the migration of foxes from one square to any other square within two "leaps"

Since this is an island, there are certain boundary conditions: The 28 squares on the water's edge have no migration possible into or out of the water for either rabbits or foxes. Similarly, there are certain initial conditions representing the starting populations of rabbits and foxes in the various squares at the time your program begins its execution.

Your job is to simulate 10 years of life on the island, using time steps of a day in length, and to determine the populations of rabbits and foxes at the end of the period in each square on the island. For each pair of rabbits in a square at the start of a birthing day, which occurs every nine weeks, a litter of babies is born. The size of the litter ranges between two and nine and varies based on both the food supply (vegetation level) and the number of rabbits in that square at the start of the day (population density), as given in Table 6.1. For each pair of foxes in a square at the start of a birthing day, which occurs every six months, a litter of kits is born. The size of the litter ranges between zero and five and varies based on both the food supply (rabbit population) and the number of foxes in that square at the start of the day (population density), as given in Table 6.2.

A fox can survive on as little as two rabbits per week, but will eat as many as four if they can be found. If the vegetation level is below 0.6, rabbits are more easily found. In that case, on any given day, there is a four in seven chance that a fox will eat a rabbit if there are sufficient rabbits available; if there are fewer rabbits than that, or if the vegetation level is at or above 0.6, the foxes will have to make do with a two in seven chance of having a meal — provided there are sufficient rabbits available at that consumption level. (If there are fewer rabbits than the

TABLE 6.1 RABBIT BIRTHS FOR PROBLEM 6-21

Vegetation at start of day	Number of rabbits at start of day				
	< 2	2 to 200	201 to 700	701 to 5000	> 5000
< 0.2	0	3	3	2	2
≥ 0.2 but < 0.5	0	4	4	3	3
≥ 0.5 but < 0.8	0	6	5	4	4
≥ 0.8	0	9	8	7	5

TABLE 6.2 RABBIT AND FOX POPULATIONS FOR PROBLEM 6-21

Rabbit population at start of day (per fox)	Number of foxes at start of day				
	< 2	2 to 10	11 to 50	51 to 100	> 100
< 3.0	0	2	2	1	0
≥3.0 but < 10	0	3	3	2	1
≥10 but < 40	0	4	3	3	2
≥40	0	5	4	3	3

number needed to keep the fox population alive, foxes that didn't get fed have a 10 percent chance that they will die off; that is, in addition to their natural death rate.) The lifespan of a fox is estimated to be four years. Use a random-number generator each day to determine whether one or more foxes die a natural death.

A rabbit consumes vegetation; each rabbit consumes 0.1 percent of the vegetation in a square per day, under non-food-constrained situations. The normal lifespan of a rabbit is estimated to be 18 months. If the vegetation level is less than 0.35, the death rate due to starvation rises dramatically, as given in Table 6.3.

TABLE 6.3 RABBIT LIFESPAN

Vegetation Level	Rabbit Lifespan
0.1 to 0.15	3 months
0.15 to 0.25	6 months
0.25 to 0.3	12 months
over 0.35	18 months

Use a random-number generator each day to determine the number of rabbits that die from a combination of starvation and natural causes. The vegetation level rises quite rapidly when not being eaten by rabbits; growing conditions are ideal on the semitropical island. The vegetation level follows the growth/consumption formula:

Vegetation at end of day =

(110% of vegetation at start of day) – (0.001 × number of rabbits at start of day)

within the limits that the vegetation level will not drop below 0.1 or grow to be more than 1.0. At the end of each day, 20 percent of the rabbit population randomly emigrates to adjoining

squares. Use a random-number generator to determine the number that actually emigrate to each of the possible adjoining squares. Similarly, since foxes range more widely, at the end of each day, every fox randomly emigrates zero, one, or two squares distant from its location at the start of that day. Note: All possible migrations are to be considered uniformly likely among the choices available.

Case 1: Uniformly, there are two foxes and 100 rabbits per square initially; the vegetation level is 1.0 everywhere.

Case 2: There are 20 foxes in one corner square and none elsewhere; there are 10 rabbits in every square except in the corner square diagonally opposite the foxes, and it contains 800 rabbits; the vegetation level is 0.3 everywhere.

Case 3: There are no foxes on the island, but there are two rabbits in each square; the initial vegetation level is 0.5 everywhere.

6-22. Develop a cellular automaton solution to a real problem and implement it.

6-23. (A research assignment) Develop the rules necessary to model the movement (erosion) of sand dunes at a beach when affected by the waves. (A similar problem is modeling the erosion of the banks of a river due to the water.)

6-24. (A research assignment) Develop the rules necessary to model the airflow across a wing as shown in Figure 6.21 (two dimensions). Select your own dimensions for the solution space and object. Select the number of grid points and write code to solve the problem.

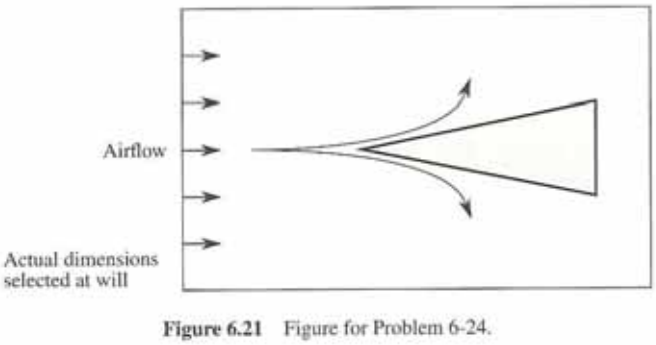


Figure 6.21 Figure for Problem 6-24.

Load Ba
Detectio

In this chap
computatio
speed. A re
termination
computatio
the various
is described

7.1 LOAD BALAN

So far, we h
in parallel.
that the pro
discussion
that some p
work is une
both situati
that would
tasks even
mentioned i
communica
there is com

Chapter 7

Load Balancing and Termination Detection

In this chapter, we introduce the concept of *load balancing*, which is used to distribute computations fairly across processors in order to obtain the highest-possible execution speed. A related issue is detecting when a computation has been completed, so-called *termination detection*. Termination detection becomes a significant issue when the computation is distributed, and it is considered here with load balancing. After developing the various load-balancing termination and detection techniques, an application example is described in detail.

7.1 LOAD BALANCING

So far, we have divided a problem into a fixed number of processes that are to be executed in parallel. Each process performs a known amount of work. In addition, it was assumed that the processes were simply distributed among the available processors without any discussion of the effects of the types of processors and their speeds. However, it may be that some processors will complete their tasks before others and become idle because the work is unevenly divided, or perhaps some processors will operate faster than others (or both situations). Ideally, we want all the processors to be operating continuously on tasks that would lead to the minimum execution time. Achieving this goal by spreading the tasks evenly across the processors is called *load balancing*. Load balancing was mentioned in Chapter 3 for the Mandelbrot calculation, in which there was no interprocess communication. Now we will develop load balancing further to include the case in which there is communication between processors. Load balancing is particularly useful when

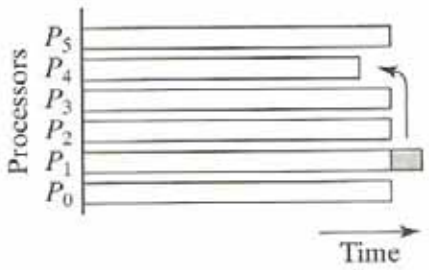
the amount of work is not known prior to execution. It also helps mitigate the effects of differences in processor speeds even when the amount of work is known in advance.

Figure 7.1 illustrates how load balancing will produce the minimum execution time (the true goal). In Figure 7.1(a), one processor, P_1, is operating for a longer period, and one processor, P_4, completes its work early. The total execution time is longer. Ideally, part of P_1's work should be given to P_4 to equalize the workload. In Figure 7.1(b), all the processors are executing for the duration of the execution, t seconds, and perfect load balancing exists. Another way of viewing this problem is that the total computation using a single processor may require k clock cycles. With p processors and no additional overhead for a parallel implementation, the execution time should be reduced to k/p clock cycles.

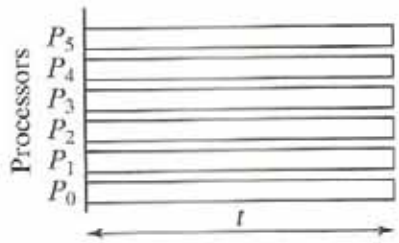
Load balancing can be attempted *statically* before the execution of any process or *dynamically* during the execution of the processes. Static load balancing is usually referred to as the *mapping problem* (Bokhari, 1981) or *scheduling problem*. Substantial literature exists on the problem, mostly using optimization techniques, starting from estimated execution times of parts of the program and their interdependencies. The following are some potential static load-balancing techniques:

- *Round robin algorithm* — passes out tasks in sequential order of processes, coming back to the first when all the processes have been given a task
- *Randomized algorithms* — selects processes at random to take tasks
- *Recursive bisection* — recursively divides the problem into subproblems of equal computational effort while minimizing message-passing
- *Simulated annealing* — an optimization technique
- *Genetic algorithm* — another optimization technique, described in Chapter 12

Figure 7.1 could also be viewed as a form of *bin packing* (i.e., placing objects into boxes to reduce the number of boxes), and scheduling can be approached with bin-packing



(a) Imperfect load balancing leading to increased execution time



(b) Perfect load balancing

Figure 7.1 Load balancing.

algorithms (Coffman, Garey, and Johnson, 1978). In our case, there would be a fixed number of equal-sized boxes (processes) and the object is to minimize the size of the boxes.

For processors/computers interconnected by a static-link interconnection network, communicating processes should be executed on processors with direct communication paths to reduce the communication delays. This is an essential part of the "mapping" problem for such systems. The solution may require different mappings for different networks. In general, there is probably no polynomial-time algorithm for solving the problem, and therefore it is regarded as a computationally intractable problem. Hence, heuristics are often used to select processors for processes.

There are several fundamental flaws with static load balancing even if a mathematical solution exists. First and foremost, it is very difficult to estimate accurately the execution times of various parts of a program without actually executing the parts. Therefore, scheduling these parts without using actual execution times is innately inaccurate. In addition, some systems may also have communication delays that vary under different circumstances, and it could be difficult to incorporate variable communication delays in static load balancing. Some problems have an indeterminate number of steps to reach their solution. For example, search algorithms commonly traverse a graph looking for the solution, and it is unknown how many paths must be searched beforehand, whether done in parallel or sequentially. Since static load balancing would not work well under these circumstances, we need to turn to dynamic load balancing.

In dynamic load balancing, all these factors are taken into account by making the division of load dependent upon the execution of the parts as they are being executed. This incurs an additional overhead during execution but is much more effective than static load balancing. In this chapter, we will concentrate upon dynamic load balancing and describe different ways that it can be achieved. We will also discuss in detail how a computation finally comes to an end, which can be a significant problem in dynamic loading balancing. This aspect is called *termination detection*.

The computation will be divided into *work* or *tasks* to be performed, and processes perform these tasks. As usual, the processes are mapped onto processors. Since our objective is to keep the processors busy, we are interested in the activity of the processors. However, we often map a single process onto each processor, so we will use the terms *process* and *processor* somewhat interchangeably.

7.2 DYNAMIC LOAD BALANCING

In dynamic load balancing, tasks are allocated to processors during the execution of the program. Dynamic load balancing can be classified as one of the following:

- Centralized
- Decentralized

In centralized dynamic load balancing, tasks are handed out from a centralized location. A clear master-slave structure exists in which a master process controls each of a set of slave processes directly. In decentralized dynamic load balancing, tasks are passed between arbitrary processes. A collection of worker processes operate upon the problem and interact

among themselves, finally reporting to a single process. A worker process may receive tasks from other worker processes and may send tasks to other worker processes to complete or pass on at their discretion.

7.2.1 Centralized Dynamic Load Balancing

In centralized dynamic load balancing, the master process holds the collection of tasks to be performed. Tasks are sent to the slave processes. When a slave process completes one task, it requests another task from the master process. This mechanism is the essential part of the so-called *work-pool* approach first introduced in Chapter 3 to generate the Mandelbrot image. In that case, the work pool held the tasks as specified by the coordinates of the pixels. The term *replicated worker* is sometimes used to describe the methodology, because all the slaves are the same. (This idea can be developed into having specialized slaves capable of performing certain tasks.) Another term used for the same methodology is *processor farm*.

The work-pool technique can be readily applied to simple divide-and-conquer problems. It can also be applied to problems in which the tasks are quite different and of different sizes. Generally, it is best to hand out the larger or most complex tasks first. If a larger task were handed out later in the computation, the slaves that completed the smaller tasks would then sit idly by waiting for the larger task to be completed.

The work-pool technique can also be readily applied when the number of tasks may change during execution. In some applications, especially search algorithms, the execution of one task may generate new tasks, though finally the number of tasks must reduce to zero, signaling that the computation is completing. A queue can be used to hold the currently waiting tasks, as shown in Figure 7.2. If all the tasks are of equal size and importance, a simple first-in-first-out queue may be acceptable. If some tasks are more important than others (e.g., are expected to lead to a solution more quickly), they would be passed to the slave processes first. Other information, such as the current best solution, may be kept by the master process.

Termination. Stopping the computation when the solution has been reached is called *termination*. A great advantage of centralized dynamic load balancing is that it is a simple matter for the master process to recognize when to terminate. For a computation in

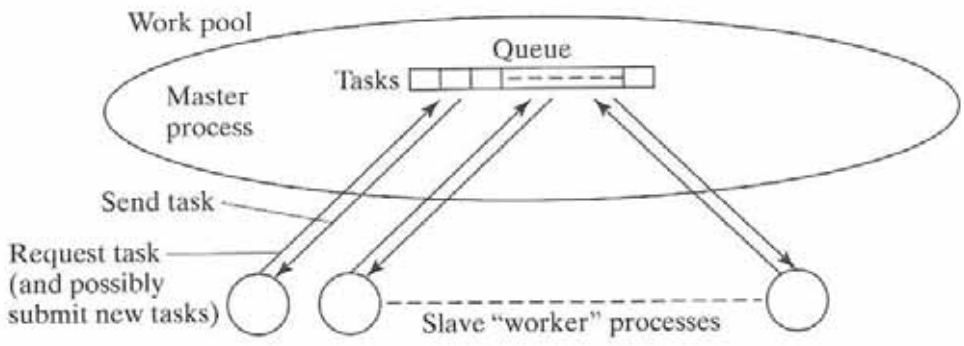


Figure 7.2 Centralized work pool.

which tasks are taken from a task queue, the computation terminates when both of the following are satisfied:

- The task queue is empty
- Every slave process is idle and has made a request for another task without any new tasks being generated

Note that it is necessary to establish that no new tasks have been generated. (Problems that do not generate new tasks during the execution, such as the Mandelbrot calculation, would terminate when the task queue is empty and all slaves have finished.)

In some applications, a slave may detect the program termination condition by some local termination condition, such as finding the item in a search algorithm. In that case, the slave process would send a termination message to the master. Then the master would close down all the other slave processes. In some applications, each slave process must reach a specific local termination condition, such as convergence on its local solutions, as in the synchronous iteration problems of Chapter 6.

7.2.2 Decentralized Dynamic Load Balancing

Although widely used, a significant disadvantage of the centralized work pool is that the master process can only issue one task at a time, and after the initial tasks have been sent, it can only respond to requests for new tasks one at a time. Thus the potential exists for a bottleneck when there are many slave processes making simultaneous requests. The centralized work pool will be satisfactory if there are few slaves and the tasks are computationally intensive. For finer-grained tasks and many slaves, it may be more appropriate to distribute the work pool into more than one site.

One approach is to distribute the work pool as shown in Figure 7.3. Here, the master has divided its initial work pool into parts and sent one part to each of a set of "mini-masters" processes (M_0 to M_{p-1}). Each mini-master controls one group of slaves. For an optimization problem, the mini-masters might find a local optimum that they would pass back the master. The master would select the best solution. It is clear that this approach

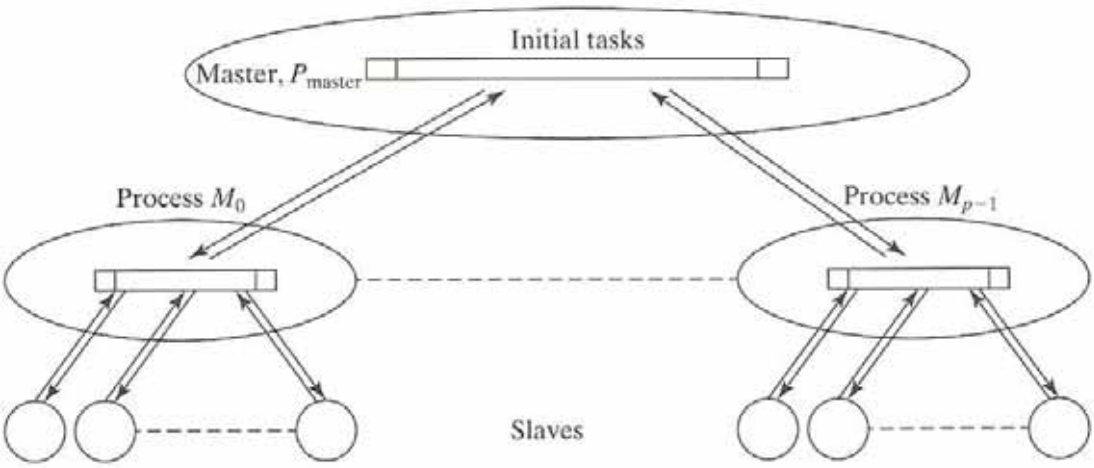


Figure 7.3 A distributed work pool.

could be developed by having several levels of decomposition; a tree could be formed with the slave processes at the leaves and internal nodes dividing up the work. This is the basic method of decomposing a task into equal subtasks. At each level in the tree, the process passes on half of the tasks to one subtree and the other half to the other subtree, assuming a binary tree. Another distributed approach would be to have the slaves actually hold a portion of the work pool and solve for this portion.

Fully Distributed Work Pool. Once the workload is distributed with processes having and generating their own tasks, the possibility exists for processes to execute tasks from each other, as illustrated in Figure 7.4. The tasks could be transferred by

1. The *receiver-initiated* method
2. The *sender-initiated* method

In the *receiver-initiated* method, a process requests tasks from other processes it selects. Typically, a process would request tasks from other processes when it has few or no tasks to perform. The method has been shown to work well at high system load. In the *sender-initiated* method, a process sends tasks to other processes it selects. Typically, in this method, a process with a heavy load passes out some of its tasks to others that are willing to accept them. This method has been shown to work well for light overall system loads. Another option is to have a mixture of both methods. Unfortunately, it can be expensive to determine process loads. In very heavy system loads, load balancing can also be difficult to achieve because of the lack of available processes.

Let us discuss load balancing in the context of the receiver-initiated method, though it can also apply to the sender-initiated method. Several strategies are feasible. Processes could be organized as a ring with a process requesting tasks from its nearest neighbors. A ring organization would suit a multiprocessor system using a ring interconnection network. Similarly, in a hypercube, processes could request tasks from those processes directly interconnected, one in each dimension. Of course, as with any strategy, one would need to be careful not to keep passing on the same task that is received.

Process Selection. Without the constraints (and advantages) of a specific interconnection network, all processes are equal candidates, and any process could select any other process. For distributed operation, each process would have its own selection algorithm, as shown in Figure 7.5. When implemented locally, this algorithm could be

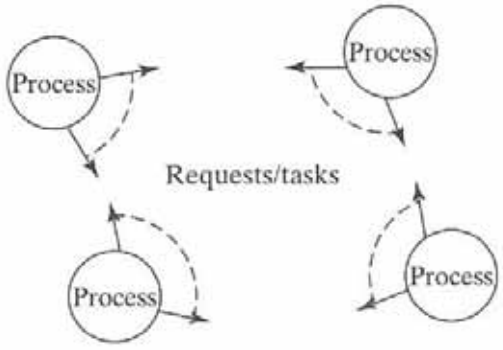


Figure 7.4 Decentralized work pool.

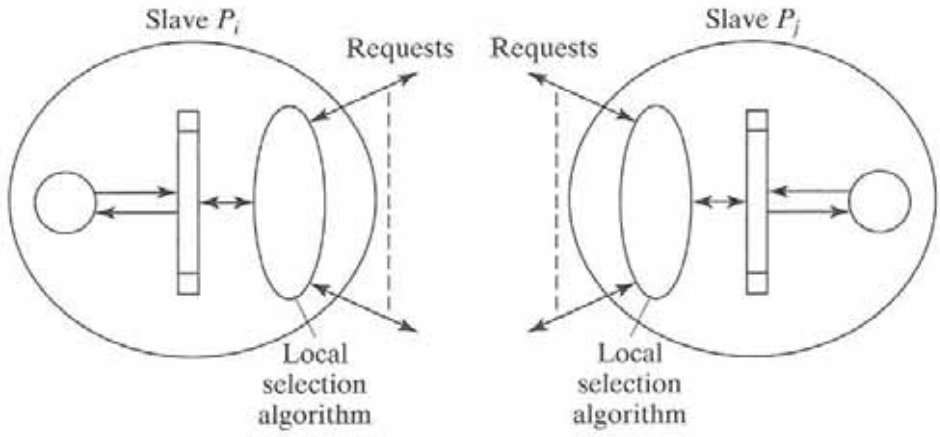


Figure 7.5 Decentralized selection algorithm requesting tasks between slaves.

applied to all the processes working on the problem or to different subsets, if the problem or network makes it desirable. Algorithms for selecting a process include the *round robin algorithm*. In the round robin algorithm, process P_i requests tasks from process P_x, where x is given by a counter that is incremented after each request, using modulo p arithmetic (p processes). If $p = 8$, x takes on the values 0, 1, 2, 3, 4, 5, 6, 7, 0, 1, 2, 3, 4, 5, 6, 7, ... The process does not select itself ($x = i$) and would cause the counter to be incremented once more when $x = i$. In the *random polling algorithm*, process P_i requests tasks from process P_x, where x is a number that is selected randomly between 0 and $p - 1$ (excluding i).

When a process receives a request for tasks, it will send a portion of the tasks it has yet to undertake to the requesting process. For example, suppose the problem is one of traversing a search tree using a depth first search. Nodes will be visited in a downward fashion from the root. A list of unvisited nodes leading from the edges of a node a process visits will be maintained. The process will select from this list a suitable set of unvisited nodes to return to the requesting process. Various strategies can be used to determine how many nodes and which nodes to return.

7.2.3 Load Balancing Using a Line Structure

Wilson (1995) describes a load-balancing technique that is particularly applicable to processors connected in a line structure (or pipeline), but the technique could be extended to other interconnection structures. He describes the technique in connection with transputers, which are often connected as a line. We consider the technique here to show the possibilities of a specific interconnection network. The basic idea is to create a queue of tasks with individual processors accessing locations in the queue, as shown in Figure 7.6. The master process (P_0 in Figure 7.6) feeds the queue with tasks at one end, and the tasks are shifted down the queue. When a worker process, P_i ($1 \le i < p$), detects a task at its input from the queue and the process is idle, it takes the task from the queue. Then the tasks to the left shuffle down the queue so that the space held by the task is filled. A new task is inserted into the left-side end of the queue. Eventually, all the processes will have a task and the queue is filled with new tasks. This mechanism will clearly keep worker processes busy. High-priority or larger tasks could be placed in the queue first.

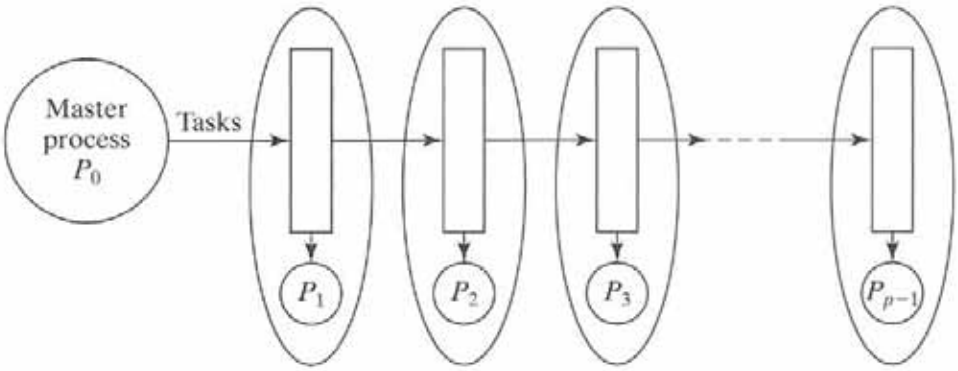


Figure 7.6 Load balancing using a pipeline structure.

The shifting actions could be orchestrated by using messages between adjacent processes. Perhaps the most elegant method is to have two processes running on each processor:

- For left and right communication
- For the current task

as shown in Figure 7.7. Three processes could also be constructed

- For left communication
- For right communication
- For the current task

These constructions are typical of transputer programs where concurrent processes are supported within the hardware of the transputer and are expected. There is no difficulty in applying the idea to some implementations of MPI that allow multiple processes on a single processor. However, some implementations of MPI do not allow multiple processes on a single processor, and in any event it might incur a very significant and possibly unacceptable overhead. (A more attractive alternative is to use threads, as described in Chapter 8.)

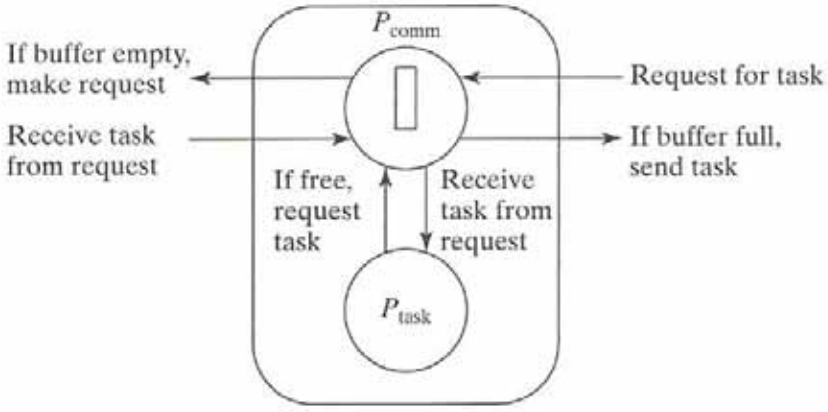


Figure 7.7 Using a communication process in line load balancing.

We could hand-code the time sharing between communication and task computation as follows:

Master process (P_0)

```
for (i = 0; i < num_tasks; i++) {
   recv(P1, request_tag);                    /* request for task */
   send(&task, P1, task_tag);                /* send tasks into queue */
}
recv(P1, request_tag);                       /* request for task */
send(&empty, P1, task_tag);                  /* end of tasks */
```

Process P_i ($1 < i < p$)

```
if (buffer == empty) {
   send(Pi-1, request_tag);                  /* request new task */
   recv(&buffer, Pi-1, task_tag);            /* task from left proc */
}
if ((buffer == full) && (!busy)) {           /* get next task */
   task = buffer;                            /* get task*/
   buffer = empty;                           /* set buffer empty */
   busy = TRUE;                              /* set process busy */
}
nrecv(Pi+1, request_tag, request);           /* check message from right */
if (request && (buffer == full)) {
   send(&buffer, Pi+1);                      /* shift task forward */
   buffer = empty;
}
if (busy) {                                  /* continue on current task */
   Do some work on task.
   If task finished, set busy to false.
}
```

A combined `sendrecv()` might be applied if available rather than a `send()`/`recv()` pair.

Nonblocking Receive Routines. In the previous code, a nonblocking `nrecv()` is necessary to check for a request being received from the right. In our pseudocode, we have simply added the parameter `request`, which is set to `TRUE` if a message has been received. In actual programming systems, specific mechanisms are present. In MPI, the nonblocking receive, `MPI_Irecv()`, returns (in a parameter) a request "handle," which is used in subsequent completion routines to wait for the message or to establish whether the message has actually been received at that point (`MPI_Wait()` and `MPI_Test()`, respectively). In effect, the nonblocking receive, `MPI_Irecv()`, posts a request for message and returns immediately.

Other Structures. Though not mentioned in Wilson (1995), it is clearly possible to extend the approach to a tree, as shown in Figure 7.8. Tasks are passed from a node into one of the two nodes below it when a node buffer becomes empty.

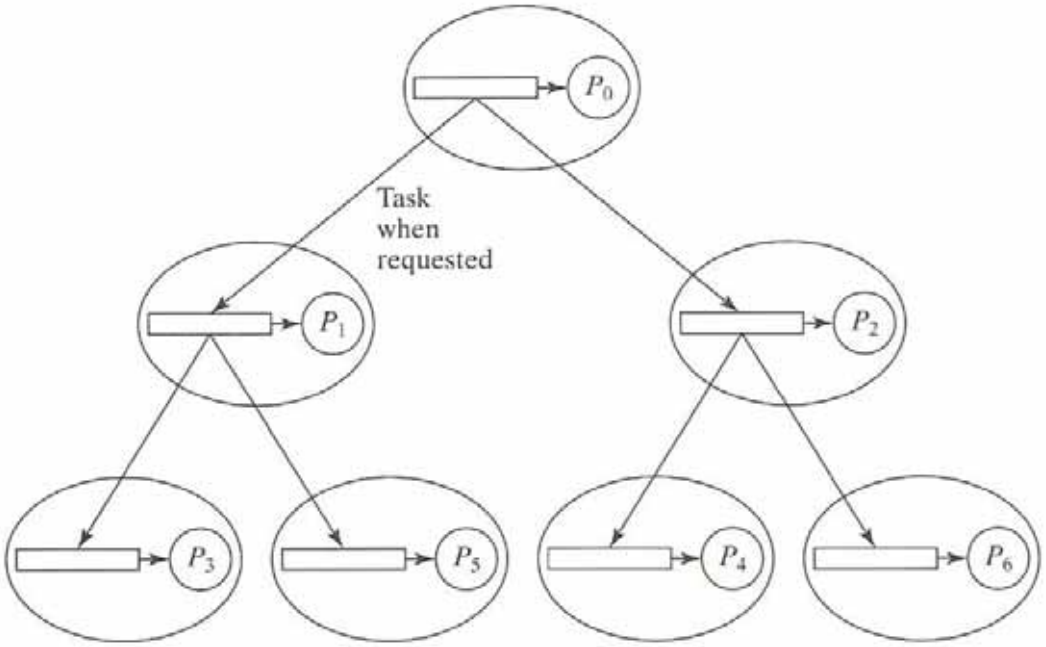


Figure 7.8 Load balancing using a tree.

7.3 DISTRIBUTED TERMINATION DETECTION ALGORITHMS

So far, we have considered distributing the tasks. Now let us look at how to terminate these distributed tasks. Various distributed termination algorithms have been proposed, but first let us examine the termination conditions.

7.3.1 Termination Conditions

When a computation is distributed, recognizing that the computation has come to an end may be difficult unless the problem is such that one process reaches a solution. In general, distributed termination at time t requires the following conditions to be satisfied (Bertsekas and Tsitsiklis, 1989):

- Application-specific local termination conditions exist throughout the collection of processes, at time t.
- There are no messages in transit between processes at time t.

The subtle difference between these termination conditions and those given for a centralized load-balancing system is the need to take into account messages in transit. The second condition is necessary for the distributed termination system because a message in transit might restart a terminated process. One could imagine a process reaching its local termination condition and terminating while a message is being sent to it from another process. The first condition is usually relatively easy to recognize. Each process can send a message to the master when its local termination conditions are satisfied. However, the second condition is more difficult to recognize. The time that it takes for messages to travel between processes will not be known in advance. One could conceivably wait a long

enough period to allow any message in transit to arrive. This approach would not be favored and would not permit portable code on different architectures.

7.3.2 Using Acknowledgment Messages

Bertsekas and Tsitsiklis (1989) describe a distributed termination method using request and acknowledgment messages. The method is very general, mathematically sound, and copes with messages being in transit when a process is about to terminate locally. Bertsekas and Tsitsiklis give formal mathematical arguments in detail.

The method is illustrated in Figure 7.9. Each process is in one of two states:

1. Inactive
2. Active

Initially, without any task to perform, the process is in the inactive state. Upon receiving a task from a process, it changes to the active state. The process that sent the task to make it enter the active state becomes its "parent." If the process passes on a task to an inactive process, it similarly becomes the parent of this process, thus creating a tree of processes, each with a unique parent. An active process could potentially receive more tasks from other active processes while it is in the active state, but these other processes are not parents of the process. Hence, the computation itself need not be a tree structure. On every occasion when a process sends a task to another process, it expects an acknowledgment message from that process. On every occasion when it receives a task from a process, it immediately sends an acknowledgment message, except if the process it receives the task from is its parent process. It only sends an acknowledgment message to its parent when it is ready to become inactive. It becomes inactive when

- Its local termination condition exists (all tasks are completed).
- It has transmitted all its acknowledgments for tasks it has received.
- It has received all its acknowledgments for tasks it has sent out.

The last condition means that a process must become inactive before its parent process. When the first process becomes idle, the computation can terminate.

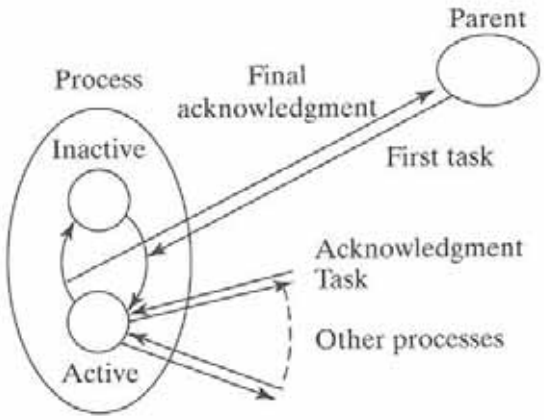


Figure 7.9 Termination using message acknowledgments.

This algorithm is perhaps the best to use because of its generality and its proven soundness. However, a particular application may lend itself to another solution, and certain interconnection structures may suggest alternative termination mechanisms.

7.3.3 Ring Termination Algorithms

For termination purposes, the processes are organized in a ring structure, as shown in Figure 7.10. The single-pass ring termination algorithm is as follows:

1. When P_0 has terminated, it generates a token that is passed to P_1.
2. When P_i $(1 \leq i < p)$ receives the token and has already terminated, it passes the token onward to P_{i+1}. Otherwise, it waits for its local termination condition and then passes the token onward. P_{p-1} passes the token to P_0.
3. When P_0 receives a token, it knows that all the processes in the ring have terminated. A message can then be sent to all the processes informing them of the global termination, if necessary.

Each process, except the first process, implements a function, as illustrated in Figure 7.11. The algorithm assumes that a process cannot be reactivated after reaching its local termination condition. This does not apply to work-pool problems in which a process can pass a new task to an idle process.

The dual-pass ring termination algorithm (Dijkstra, Feijen, and Gasteren, 1983) can handle processes being reactivated but requires two passes around the ring. The reason for reactivation is for process P_i to pass a task to P_j where $j < i$ and after a token has passed P_j, as shown in Figure 7.12. If this occurs, the token must recirculate through the ring a second time. To differentiate these circumstances, tokens are colored white or black. Processes are also colored white or black. Receiving a black token means that global termination may not

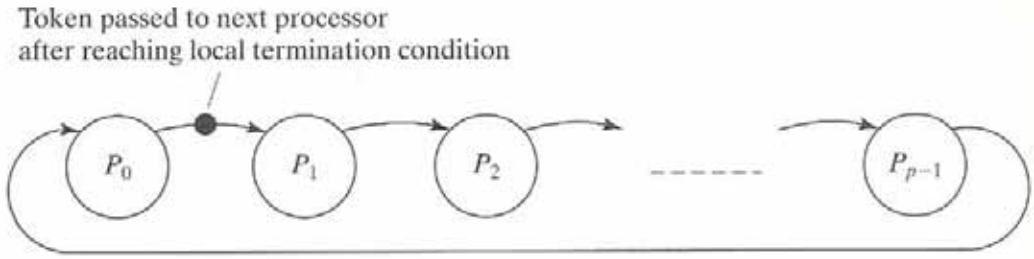


Figure 7.10 Ring termination detection algorithm.

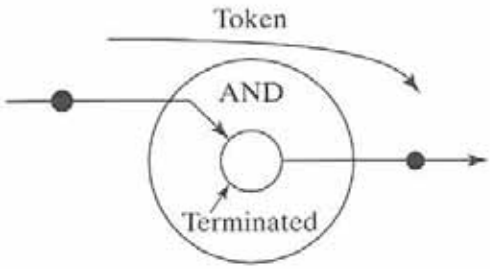


Figure 7.11 Process algorithm for local termination.

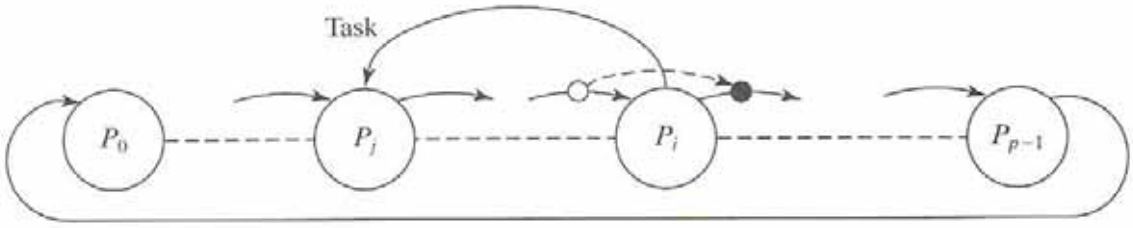


Figure 7.12 Passing task to previous processes.

have occurred and the token must be recirculated around the ring again. The algorithm is as follows, again starting at P_0:

1. P_0 becomes white when it has terminated and generates a white token to P_1.
2. The token is passed through the ring from one process to the next when each process has terminated, but the color of the token may be changed. If P_i passes a task to P_j where $j < i$ (i.e., before this process in the ring), it becomes a *black process*; otherwise it is a *white process*. A black process will color a token black and pass it on. A white process will pass on the token in its original color (either black or white). After P_i has passed on a token, it becomes a white process. P_{p-1} passes the token to P_0.
3. When P_0 receives a black token, it passes on a white token; if it receives a white token, all processes have terminated.

Note that in both ring algorithms, P_0 becomes the central point for global termination. Also, it is assumed that an acknowledge signal is generated to each request.

Tree Algorithm. The local actions described in Figure 7.11 can be applied to various interconnection structures, notably a tree structure, to indicate that processes up to that point have terminated. Two branches of a tree using this mechanism are shown in Figure 7.13. Now a token is passed forward when the tokens are received from each branch of the tree and the local termination condition exists. When the root receives its

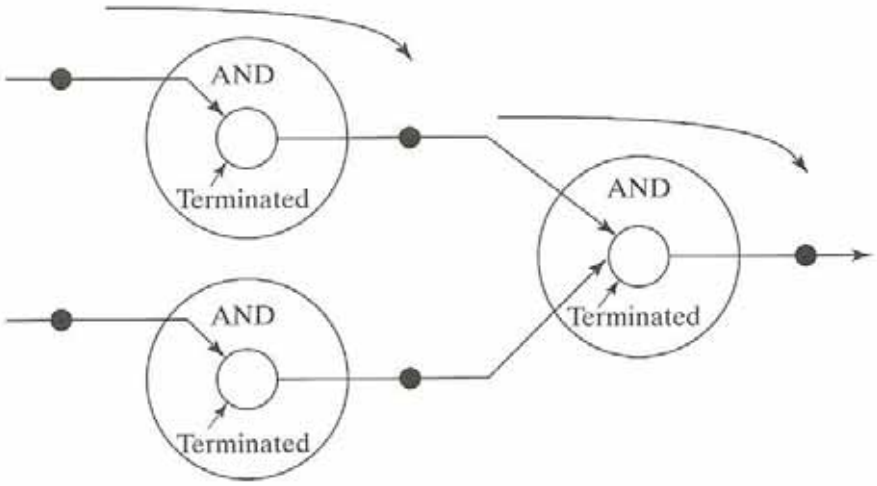


Figure 7.13 Tree termination.

full complement of tokens and has terminated, global termination has occurred. Again all other processes must then be informed, perhaps by a tree broadcast algorithm.

7.3.4 Fixed Energy Distributed Termination Algorithm

Another termination algorithm uses the notation of a fixed quantity within the system, colorfully termed "energy." This energy is similar to a token but has a numeric value. The system starts with all the energy being held by one process, the master process. The master process passes out portions of the energy with the tasks to processes making requests for tasks. Similarly, if these processes receive requests for tasks, the energy is divided further and passed to them. When a process becomes idle, it passes the energy it holds back before requesting a new task. This energy could be passed directly back to the master process or to the process giving it the original task. In the latter case, the algorithm will create a treelike structure, and a process will not hand back its energy until all the energy it has handed out is returned and combined to the total energy held. When all the energy is returned to the root and the root becomes idle, all the processes must be idle and the computation can terminate.

A significant disadvantage of the fixed energy method is that dividing the energy will be of finite precision and adding the partial energies may not equate to the original energy if floating-point arithmetic is used. In addition, one can only divide the energy so far before it becomes essentially zero. Integer arithmetic with verification can generally overcome the first problem if the original integer energy is large enough to cope with the number of divisions.

7.4 PROGRAM EXAMPLE

In this section, we will discuss how the various load-balancing strategies can be applied to a representative problem. There are several application areas, including the obvious search and optimization areas. Other areas include image processing, ray tracing, and volume rendering. In fact, any problem that can be divided and conquered is a candidate for a work-pool approach. For the most part, problems that can take greatest advantage of dynamic load balancing are those in which the number of tasks is variable and unknown. Of course, dynamic load balancing is also very advantageous to a heterogeneous network of computers.

7.4.1 Shortest-Path Problem

We will investigate the problem of finding the shortest distance between two points on a graph. This is a very well known problem appearing in some form in most sequential programming classes. It can be stated as follows:

> Given a set of interconnected nodes where the links between the nodes are marked with "weights," find the path from one specific node to another specific node that has the smallest accumulated weights.

The interconnected nodes can be described by a *graph*. In graph terminology, the nodes are called *vertices*, and the links are called *edges*. If the edges have implied directions, that is, if an edge can only be traversed in one direction, the graph is a *directed graph*. The problem

is one of searching for the best path through the graph. The graph itself could be used to find the solution to many different problems; for example,

1. The shortest distance between two towns or other points on a map, where the weights represent distance
2. The quickest route to travel, where the weights represent time (the quickest route may not be the shortest route if different modes of travel are available; for example, flying to certain towns)
3. The least expensive way to travel by air, where the weights represent the cost of the flights between cities (the vertices)
4. The best way to climb a mountain given a terrain map with contours
5. The best route through a computer network for minimum message delay (the vertices represent computers, and the weights represent the delay between two computers)
6. The most efficient manufacturing system, where the weights represent hours of work

"The best way to climb a mountain" will be used as an example, as illustrated in Figure 7.14. The corresponding graph is shown in Figure 7.15, where the weights indicate the amount of effort that would be expended in traversing the route between two connected camp sites. Note in this example that the graph is a directed graph and the weights are associated with traversing the path in a particular direction. Theoretically, we should make paths between all the camps in both directions, an exhaustively connected graph, though it would still be a directed graph since the weights would be different in each direction. The effort in one direction may be different from the effort in the opposite direction (downhill instead of uphill!). In some problems, the weights would be the same in both directions. For example, in finding the shortest route to drive, the distance is the same in both directions. The weights would be the same, an *undirected graph*.

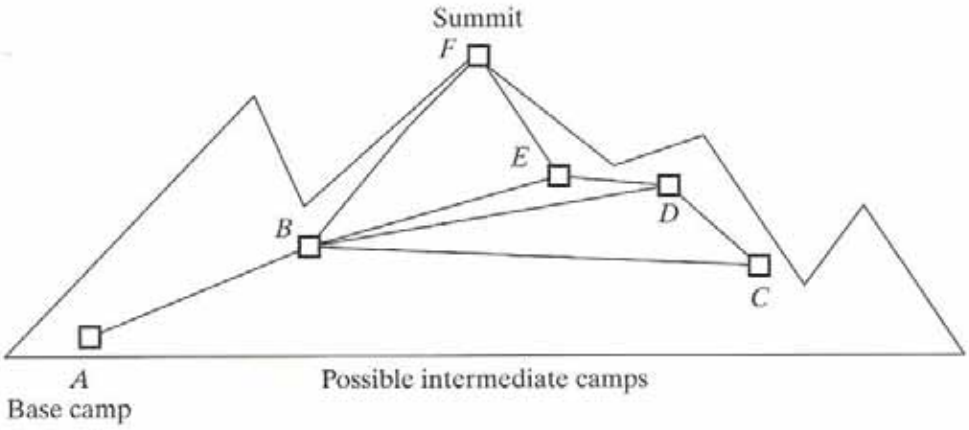


Figure 7.14 Climbing a mountain.

7.4.2 Graph Representation

We first need to establish the way that the graph is to be represented in the program. As will be familiar from sequential programming, there are two basic ways that a graph can be represented in a program:

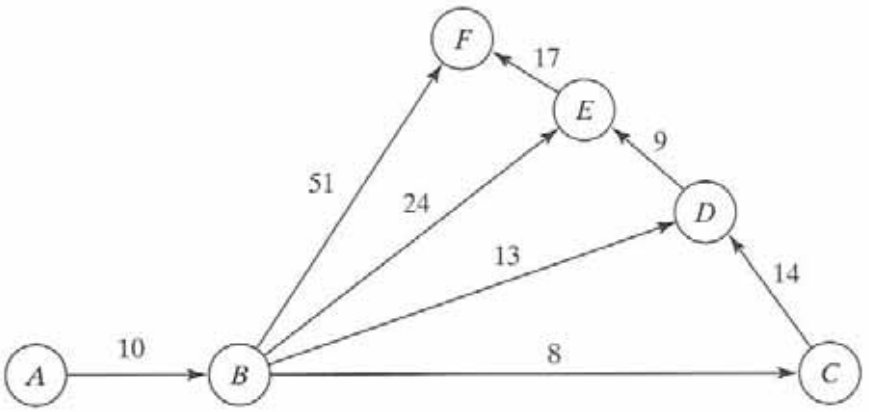


Figure 7.15 Graph of mountain climb.

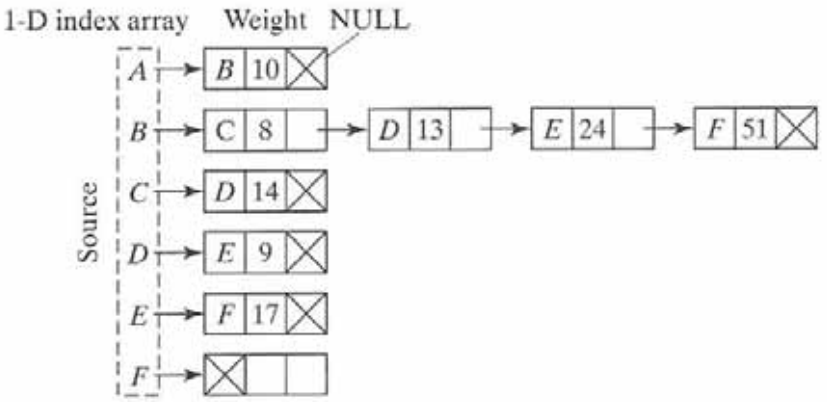
1. Adjacency matrix — a two-dimensional array, `a`, in which `a[i][j]` holds the weight associated with the edge between vertex *i* and vertex *j* if one exists
2. Adjacency list — for each vertex, a list of vertices directly connected to the vertex by an edge and the corresponding weights associated with the edges

Both methods are shown in Figure 7.16 for our mountain-climbing problem. The adjacency list is implemented as a linked list. The order of the edges in the adjacency list is arbitrary.

Destination

Source	A	B	C	D	E	F
A	∞	10	∞	∞	∞	∞
B	∞	∞	8	13	24	51
C	∞	∞	∞	14	∞	∞
D	∞	∞	∞	∞	9	∞
E	∞	∞	∞	∞	∞	17
F	∞	∞	∞	∞	∞	∞

(a) Adjacency matrix



(b) Adjacency list

Figure 7.16 Representing a graph.

The method chosen will depend upon characteristics of the graph and the structure of the program. For sequential programs, the adjacency matrix is normally used for dense graphs—graphs where there are many edges from each vertex. The adjacency list is used for sparse graphs—graphs where there are few edges from each vertex. The difference is based upon space (storage) requirements. An adjacency matrix has an $O(v^2)$ space requirement, and an adjacency list has an $O(ve)$ space requirement, where there are e edges from each vertex and v vertices in all. In general, e will be different for each vertex, and therefore the upper bound on the space requirement of an adjacency list is given by $O(ve_{max})$. Accessing the adjacency list is regarded as slower than accessing the adjacency matrix, as it requires the linked list to be traversed sequentially, which potentially requires v steps. For parallel programs, an adjacency list could be accessed in parallel to speed up the process. In addition to space and time characteristics, for parallel programs we need to consider the partitioning of tasks and its effect on accessing the information. For now, let us assume an adjacency matrix representation (even though our graph is sparse).

7.4.3 Searching a Graph

In our example, the search to the summit is quite simple because there are only a few ways to the summit. But in more complex problems, the search is not so manageable, and an algorithmic approach is necessary. Single-source shortest-path graph algorithms find the minimum accumulation of weights from a source vertex to a destination vertex. Two well-known single-source shortest-path algorithms are candidates for identifying the best way to the summit:

- Moore's single-source shortest-path algorithm (Moore, 1957)
- Dijkstra's single-source shortest-path algorithm (Dijkstra, 1959)

The two algorithms are similar. Moore's is chosen because it is more amenable to parallel implementation, although it may do more work (Adamson and Tick, 1992). The weights must all be positive values for the algorithm to work. (Other algorithms exist that will work with both positive and negative weights.)

Moore's Algorithm. Starting with the source vertex, the basic algorithm implemented when vertex i is being considered is as follows: Find the distance to vertex j through vertex i and compare with the current minimum distance to vertex j. Change the minimum distance if the distance through vertex i is shorter. In mathematical notation, if d_i is the current minimum distance from the source vertex to vertex i, and $w_{i,j}$ is the weight of the edge from vertex i to vertex j, we have

$$d_j = \min(d_j, d_i + w_{i,j})$$

The algorithm is illustrated in Figure 7.17. Interestingly, the problem could be solved by simply applying the preceding formula repeatedly (an iterative solution). See Bertsekas and Tsitsiklis (1989) for further details.

The formula here is implemented using a directed search. A first-in-first-out vertex queue is created and holds a list of vertices to examine. Vertices are considered only when they are in the vertex queue. Initially, only the source vertex is in the queue. Another

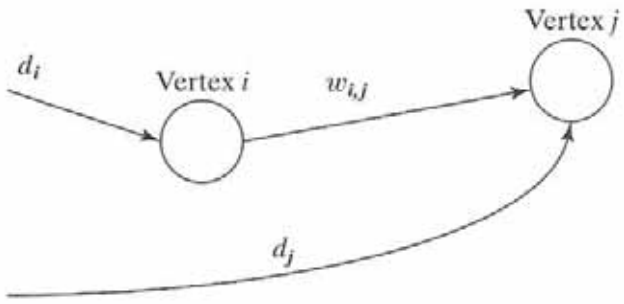


Figure 7.17 Moore's shortest-path algorithm.

structure is needed to hold the current shortest distance from the source vertex to each of the other vertices. Suppose there are n vertices, and vertex 0 is the source vertex. The current shortest distance from the source vertex to vertex i will be stored in the array `dist[i]` ($1 \le i < n$). At first, none of these distances will be known and the array elements are initialized to infinity. Suppose `w[i][j]` holds the weight of the edge from vertex i and vertex j (infinity if no edge). The code could be of the form

```
newdist_j = dist[i] + w[i][j];
if (newdist_j < dist[j]) dist[j] = newdist_j;
```

When a shorter distance is found to vertex j, vertex j is added to the queue (if not already in the queue), which will cause vertex j to be examined again. This is an important aspect of this algorithm, which is not present in Dijkstra's algorithm.

Stages in Searching a Graph. To see how this algorithm proceeds from the source vertex, let us follow the steps using our mountain-climbing graph as the example.

The initial values of the two key data structures are

The element `dist[A]` will always be zero when A is the source, but the structure provides for complete generality should a vertex other than A be selected as the source vertex.

First, each of the edges emanating from vertex A is examined. In our graph, that will be vertex B. The weight to vertex B is 10, which will provide the first (and actually the only distance) to vertex B. Both data structures, `vertex_queue` and `dist[]`, are updated as follows:

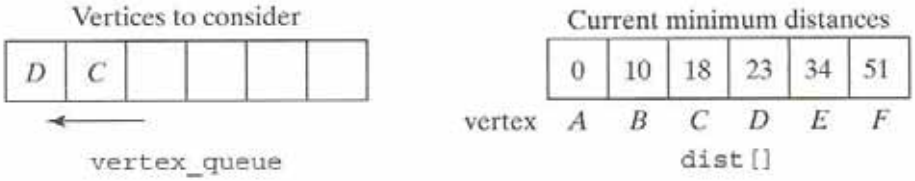
Once a new vertex, B, is placed in the vertex queue, the task of searching around vertex B begins. Now we have four edges to examine: to C, to D, to E, and to F. In this

algorithm, it is not necessary to examine these edges in any specific order. Dijkstra's algorithm requires the nearest vertex to be examined first, which imposes sequential processing. However, Moore's algorithm may require vertices to be reexamined. To demonstrate, let us examine the edges in the order *F*, *E*, *D*, and *C*.

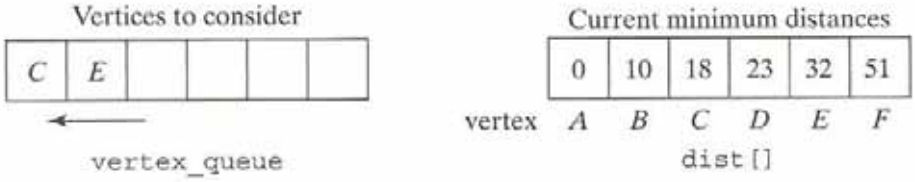
The distances through vertex *B* to the vertices are `dist[F]` = 10 + 51 = 61, `dist[E]` = 10 + 24 = 34, `dist[D]` = 10 + 13 = 23, and `dist[C]` = 10 + 8 = 18. Since all were new distances, all the vertices are added to the queue (except *F*), as follows:

Vertex *F* need not be added because it is the destination with no outgoing edges and requires no processing. (If *F* were added, it would be discovered that there were no outgoing edges.)

Starting with vertex *E*, which has one edge to vertex *F* with a weight of 17, the distance to vertex *F* through vertex *E* is `dist[E]` + 17 = 34 + 17 = 51, which is less than the current distance to vertex *F* and replaces this distance, leading to

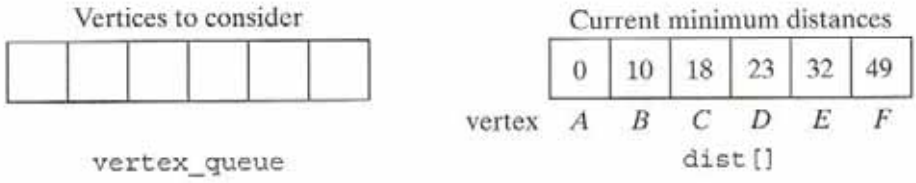


Next is vertex *D*. There is one edge to vertex *E* with the weight of 9, giving the distance to vertex *E* through vertex *D* of `dist[D]` + 9 = 23 + 9 = 32, which is less than the current distance to vertex *E* and replaces this distance. Vertex *E* is added to the queue, as follows:



Next is vertex *C*. We have one edge to vertex *D* with a weight of 14. Hence, the (current) distance through vertex *C* to vertex *D* is `dist[C]` + 14 = 18 + 14 = 32. This is greater than the current distance to vertex *D* of 23, so this distance is left stored.

Next is vertex *E* (again). There is one edge to vertex *F* with the weight of 17, giving the distance to vertex *F* through vertex *E* of `dist[E]` + 17 = 32 + 17 = 49, which is less than the current distance to vertex *F* and replaces this distance, as follows:



There are no more vertices to consider. We have the minimum distance from vertex A to each of the other vertices, including the destination vertex, F. Usually, the actual path is also required in addition to the distance. Then the path needs to be stored as the distances are recorded. The path in our case is $A \rightarrow B \rightarrow D \rightarrow E \rightarrow F$.

Sequential Code. The specific details of maintaining the vertex queue are omitted. Let `next_vertex()` return the next vertex from the vertex queue, or `num_vertex` if none. We will assume that an adjacency matrix is used, named `w[][]`, which is accessed sequentially to find the next edge. The sequential code could then be of the form

```
while ((i = next_vertex()) != num_vertex)          /* while a vertex */
   for (j = 0; j < n; j++)                          /* get next edge */
      if (w[i][j] != infinity) {                    /* if an edge */
         newdist_j = dist[i] + w[i][j];
         if (newdist_j < dist[j]) {
            dist[j] = newdist_j;
            append_queue(j);            /* vertex to queue if not there */
         }
      }                                 /* no more vertices to consider */
```

Parallel Implementations. We will look at both the centralized work pool and decentralized work pool solutions.

Centralized Work Pool. The first parallel implementation to consider uses a centralized work pool holding the vertex queue, `vertex_queue[]`, as tasks. Each slave takes vertices from the vertex queue and returns new vertices in the manner illustrated previously in Figure 7.2. For the slaves to identify edges and compute distances, they need access to both the structure holding the graph weights (adjacency matrix or adjacency list) and the array holding the current minimum distances, `dist[]`. If this information is held by the master process, messages will need to be sent to the master to access the information. This could lead to a very significant communication overhead. Since the structure holding the graph weights is fixed, this structure could be copied into each slave. We will assume a copied adjacency matrix. For now, let us assume that the distance array, `dist[]`, is held centrally and simply copied with the vertex in its entirety. Instead, individual requests for distances could also be made. The code could be of the form

Master

```
while (vertex_queue() != empty) {
   recv(P_ANY, source = P_i);                  /* request task from slave */
   v = get_vertex_queue();
   send(&v, P_i);                              /* send next vertex and */
   send(&dist, &n, P_i);                       /* current dist array */
   recv(&j, &dist[j], P_ANY, source = P_i);    /* new distance received */
   append_queue(j, dist[j]);                   /* append vertex to queue */
                                               /* and update distance array */
};
recv(P_ANY, source = P_i);                     /* request task from slave */
send(P_i, termination_tag);                    /* termination message*/
```

Slave (process i)

```
send(P_master);                                  /* send request for task */
recv(&v, P_master, tag);                         /* get vertex number */
if (tag != termination_tag) {
   recv(&dist, &n, P_master);                    /* and dist array */
   for (j = 0; j < n; j++)                       /* get next edge */
      if (w[v][j] != infinity) {                 /* if an edge */
         newdist_j = dist[v] + w[v][j];
         if (newdist_j < dist[j]) {
            dist[j] = newdist_j;
            send(&j, &dist[j], P_master);        /* add vertex to queue */
         }                                       /* send updated distance */
      }
}
```

Clearly, the vertex number and distance array could be sent in one message. Note too that individual slaves may have distances that are not exactly the same because they are being updated continually by different slaves.

The master waits for requests from any slave but must respond to the specific slave that makes a request. In our pseudocode notation, `source = P_i` is used to indicate the source of the message. In an actual programming system, the source could be identified by making each slave send its identification (possibly as a unique tag). In the case of MPI, the actual source of the message can be found by reading the status word returned by the `MPI_Recv()` routine.

Decentralized Work Pool. One of the distributed work-pool approaches can be applied to our problem. The task queue, in our case `vertex_queue[]`, could also be distributed. A convenient approach is to assign slave process i to search around vertex i only and for it to have the vertex queue entry for vertex i if this exists in the queue. In other words, one element of the queue is reserved specifically to hold vertex i, and this entry is in process i. The array `dist[]` will also be distributed among the processes so that process i maintains the current minimum distance to vertex i. Process i also stores an adjacency matrix/list for vertex i, for the purpose of identifying the edges from vertex i.

With our arrangements, the algorithm can proceed as follows: The search will be activated by a coordinating process loading the source vertex into the appropriate process. In our case, vertex A is the first vertex to search. The process assigned to vertex A is activated. This process will immediately begin searching around its vertex to find distances to connected vertices. The distances will then be sent to the appropriate processes. The distance to vertex j will be sent to process j to be compared with its currently stored value and replaced if the currently stored value is larger. In our case, the process responsible for vertex B will be contacted with the distance to vertex B. In this fashion, all minimum distances will be updated during the search. If the contents of `d[i]` changes, process i will be reactivated to search again. Figure 7.18 shows the message-passing. Message-passing will distribute across many of the slave processes, rather than be focused on the master process.

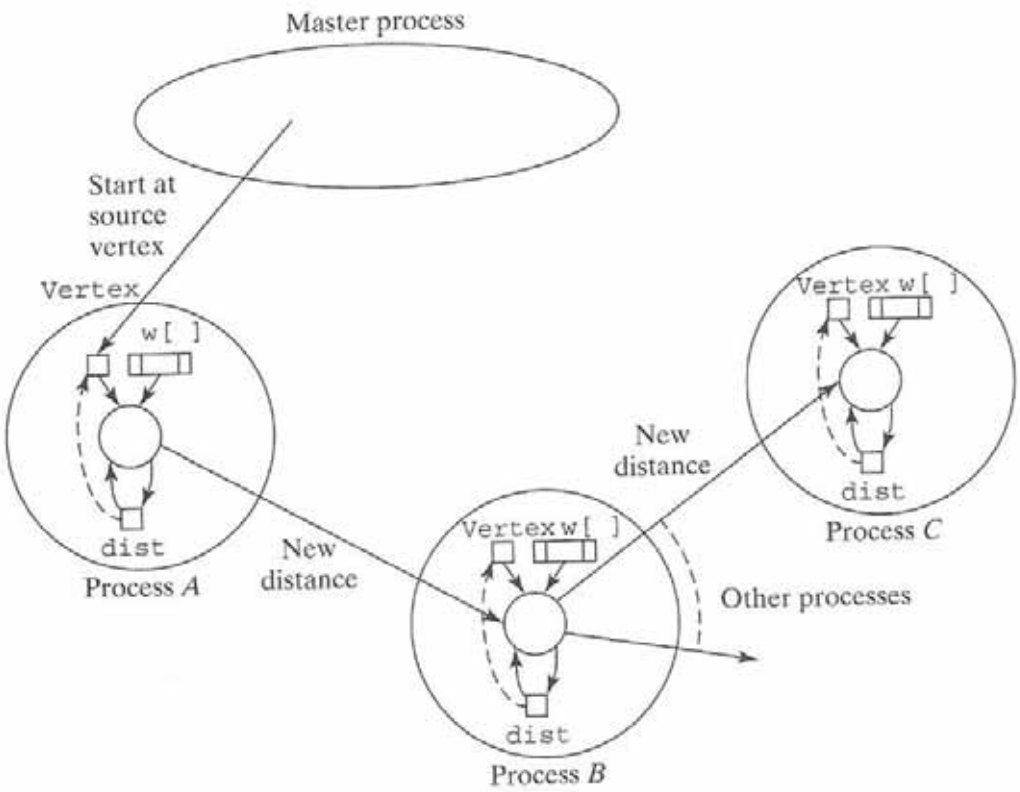


Figure 7.18 Distributed graph search.

A code segment for the slave processes might take the form

Slave (process *i*)

```
recv(newdist, P_ANY);
if (newdist < dist) {
   dist = newdist;
   vertex_queue = TRUE;                    /* add to queue */
} else vertex_queue == FALSE;
if (vertex_queue == TRUE)                  /* start searching around vertex */
   for (j = 0; j < n; j++)                 /* get next edge */
      if (w[j] != infinity) {
         d = dist + w[j];
         send(&d, P_j);                    /* send distance to proc j */
      }
```

This could certainly be simplified to:

Slave (process *i*)

```
recv(newdist, P_ANY);
if (newdist < dist) {
   dist = newdist;                         /* start searching around vertex */
   for (j = 1; j < n; j++)                 /* get next edge */
      if (w[j] != infinity) {
         d = dist + w[j];
         send(&d, P_j);                    /* send distance to proc j */
      }
}
```

A mechanism is necessary to repeat the actions and terminate when all the processes are idle. The mechanism must cope with messages in transit. The simplest solution is to use synchronous message-passing, in which a process cannot proceed until the destination has received the message. It is left as an exercise to investigate this method and the more powerful method of identifying the unique parent that receives an acknowledgment last, as described in Section 7.3.

Note that a process is only active after its vertex is placed on the queue, and it is possible for many processes to be inactive, leading to an inefficient solution. The method is also impractical for a large graph if one vertex is allocated to each processor. In that case, a group of vertices could be allocated to each processor.

7.5 SUMMARY

This chapter introduced the following:

- Centralized and distributed work pools and load-balancing techniques
- Several distributed termination algorithms
- Shortest-path graph-searching application

FURTHER READING

There have been a large number of research papers on static and dynamic scheduling of tasks over the years. Static load balancing is found in Graham (1972). Another early task-allocation paper is Chu et al. (1980), with references to previous work. Heuristic methods are described in Efe (1982), Lo (1988), and Shirazi and Wang (1990). Static and dynamic methods are compared in Iqbal, Salz, and Bokhari (1986). Other details and methods of static load balancing can be found in Lewis and El-Rewini (1992) and El-Rewini (1996). Textbooks solely on scheduling include Bharadwaj et al. (1996), which provides a detailed mathematical treatment.

Load balancing in distributed systems is also described in many papers; for example, Tantawi and Towsley (1985), Shivaratri, Krueger, and Singhal (1992), and El-Rewini, Ali, and Lewis (1995). A collection of papers is published in Shirazi, Hurson, and Kavi (1995). Jacob (1996) considers load balancing specifically in a network of workstations. The powerful dynamic load-balancing technique using the concept of a single parent that receives an acknowledgment last is described fully, with its mathematical underpinning, in Bertsekas and Tsitsiklis (1989). The method is also used in parallel programs in Lester (1993). The concept of viewing load balancing as a physical system optimizing for minimum energy is described in Fox et al. (1988). Termination detection is treated in Barbosa (1996).

Mateti and Deo (1982), Paige (1985), and Adamson and Tick (1992) consider parallel algorithms for the shortest-path problem. Lester (1993) considers parallel programming aspects of the shortest-path problem.

BIBLIOGRAPHY

ADAMSON, P., AND E. TICK (1992), "Parallel Algorithms for the Single-Source Shortest-Path Problem," *Proc. 1992 Int. Conf. Par. Proc.*, Vol. 3, pp. 346–350.

BARBOSA, V. C. (1996), *An Introduction to Distributed Algorithms*, MIT Press, Cambridge, MA.

BERMAN, K. A., AND J. L. PAUL (1997), *Fundamentals of Sequential and Parallel Algorithms*, PWS, Boston, MA.

BERTSEKAS, D. P., AND J. N. TSITSIKLIS (1989), *Parallel and Distributed Computation Numerical Methods*, Prentice Hall, Englewood Cliffs, NJ.

BHARADWAJ, V., D. GHOSE, V. MANI, AND T. G. ROBERTAZZI (1996), *Scheduling Divisible Loads in Parallel and Distributed Systems*, IEEE CS Press, Los Alamitos, CA.

BOKHARI, S. H. (1981), "On the Mapping Problem," *IEEE Trans. Comput.*, Vol. C-30, No. 3, pp. 207–214.

CHU, W. W., L. J. HOLLOWAY, M.-T. LAN, AND K. EFE (1980), "Task Allocation in Distributed Data Processing," *Computer*, Vol. 13, No. 11, pp. 57–69.

COFFMAN, E. G., JR., M. R. GAREY, AND D. S. JOHNSON (1978), "Application of Bin-Packing to Multiprocessor Scheduling," *SIAM J. on Computing*, Vol. 7, No. 1, pp. 1–17.

DIJKSTRA, E. W. (1959), "A Note on Two Problems in Connexion with Graphs," *Numerische Mathematik*, Vol. 1, pp. 269–271.

DIJKSTRA, E. W., W. H. FEIJEN, AND A. J. M. V. GASTEREN (1983), "Derivation of a Termination Detection Algorithm for a Distributed Computation," *Information Processing Letters*, Vol. 16, No. 5, pp. 217–219.

EFE, K. (1982), "Heuristic Models of Task Assignment Scheduling in Distributed Systems," *Computer*, Vol. 15, No. 6, pp. 50–56.

EL-REWINI, H. (1996), "Partitioning and Scheduling," Chap. 9 in *Parallel and Distributed Computing Handbook*, Zomaya, A. Y., ed., McGraw-Hill, NY.

EL-REWINI, H., H. H. ALI, AND T. LEWIS (1995), "Task Scheduling in Multiprocessor Systems," *Computer*, Vol. 28, No. 12, pp. 27–37.

FOX, G., M. JOHNSON, G. LYZENGA, S. OTTO, J. SALMON, AND D. WALKER (1988), *Solving Problems on Concurrent Processors*, Volume 1, Prentice Hall, Englewood Cliffs, NJ.

GRAHAM, R. L. (1972), "Bounds on Multiprocessing Anomalies and Packing Algorithms," *Proc. AFIPS 1972 Spring Joint Computer Conference*, pp. 205–217.

IQBAL, M. A., J. H. SALZ, AND S. H. BOKHARI (1986), "A Comparative Analysis of Static and Dynamic Load Balancing Strategies," *Proc. 1986 Int. Conf. Par. Proc.*, pp. 1040–1047.

JACOB, J. C. (1996), "Task Spreading and Shrinking on a Network of Workstations with Various Edge Classes," *Proc. 1996 Int. Conf. Par. Proc.*, Part III, pp. 174–181.

LESTER, B. (1993), *The Art of Parallel Programming*, Prentice Hall, Englewood Cliffs, NJ.

LEWIS, T. G., AND H. EL-REWINI (1992), *Introduction to Parallel Computing*, Prentice Hall, Englewood Cliffs, NJ.

LO, V. M. (1988), "Heuristic Algorithms for Task Assignment in Distributed Systems," *IEEE Trans. Comput.*, Vol. 37, No. 11, pp. 1384–1397.

LO, V. M., AND S. RAJOPADHYE (1990), "Mapping Divide-and-Conquer Algorithms to Parallel Architectures," *Proc. 1990 Int. Conf. Par. Proc.*, Part III, pp. 128–135.

MATETI, P., AND N. DEO (1982), "Parallel Algorithms for the Single Source Shortest Path Problem," *Computing*, Vol. 29, pp. 31–49.

MATTSON, T. G. (1996), "Scientific Computation," Chap. 34 in *Parallel and Distributed Computing Handbook*, Zomaya, A. Y., ed., McGraw-Hill, NY.

MOORE. E. F. (1957), "The Shortest Path Through a Maze," *Proc. Int. Symp. on Theory of Switching Circuits*, pp. 285–292.

PAIGE, R. C. (1985), "Parallel Algorithms for Shortest Path Problems," *Proc. 1985 Int. Conf. Par. Proc.*, pp. 14–20.

SHIRAZI, B., AND M. WANG (1990), "Analysis and Evaluation of Heuristic Methods for Static Task Scheduling," *J. Par. Dist. Comput.*, Vol. 10, pp. 222–232.

SHIRAZI, B. A., A. R. HURSON, AND K. M. KAVI (1995), *Scheduling and Load Balancing in Parallel and Distributed Systems*, IEEE CS Press, Los Alamitos, CA.

SHIVARATRI, N. G., P. KRUEGER, AND M. SINGHAL (1992), "Load Distribution for Locally Distributed Systems," *Computer*, Vol. 25, No. 12, pp. 33–44.

TANTAWI, A. N., AND D. TOWSLEY (1985), "Optimal Load Balancing in Distributed Computer Systems," *J. ACM*, Vol. 32, No. 2, pp. 445–465.

WILSON, G. V. (1995), *Practical Parallel Programming*, MIT Press, Cambridge, MA.

ZOMAYA, A. Y., ed. (1996), *Parallel and Distributed Computing Handbook*, McGraw-Hill, NY.

PROBLEMS

Scientific/Numerical

7-1. One approach for assigning processes to processors is to make the assignment random using a random-number generator. Investigate this technique by applying it to a parallel program that adds together a sequence of numbers.

7-2. Write a parallel program that will implement the load-balancing technique using the pipeline structure described in Section 7.2.3 for any arbitrary set of independent arithmetic tasks.

7-3. The traveling salesperson problem is a classical computer science problem (though it might also be regarded as a real-life problem). Starting at one city, the objective is to visit each of n cities exactly once and return to the first city on a route that minimizes the distance traveled. The n cities can be regarded as variously connected. The connections can be described by a weighted graph. Write a parallel program to solve the traveling salesperson problem with real data obtained from a map to include 25 major cities.

7-4. Implement Moore's algorithm using the load-balancing line structure described in Section 7.2.3.

7-5. As noted in the text, the decentralized work-pool approach described in Section 7.4 for searching a graph is inefficient in that processes are only active after their vertex is placed on the queue. Develop a more efficient work-pool approach that keeps processes more active.

7-6. Write a load-balancing program using Moore's algorithm and a load-balancing program using Dijkstra's algorithm for searching a graph. Compare the performance of each algorithm and make conclusions.

Real Life

7-7. Single-source shortest-path algorithms can be used to find the shortest route for messages through any interconnection network one would like to devise. Write a parallel program that will find all the shortest routes through an arbitrary interconnection network and the specific one of your computer cluster if not fully switched.

7-8. Quality-of-service (QOS) describes how well a communication network, most notably the Internet, can provide data transfers within constraints. There may be several parameters (initial response time, maximum data transmission delay, etc.) that could be specified by the user and could be modeled by separate weights on each arc. Write a parallel program that can search a graph in which each arc has two weights and attempts to find a path which minimizes both accumulated weights. It may not be possible to obtain the absolute minimum of both accumulated weights, and one may need to provide an acceptable maximum value for each of the accumulated weights.

7-9. You have been commissioned to develop a challenging maze to be constructed at a stately home. The maze is to be laid out on a grid such as shown in Figure 7.19. Develop a parallel program that will find the positions of the hedges that result in the *longest time* in the maze if one uses the maze algorithm "Keep to the path where there is a hedge or wall on the left," as illustrated in Figure 7.19, which is guaranteed to find the exit eventually (Berman and Paul, 1997).

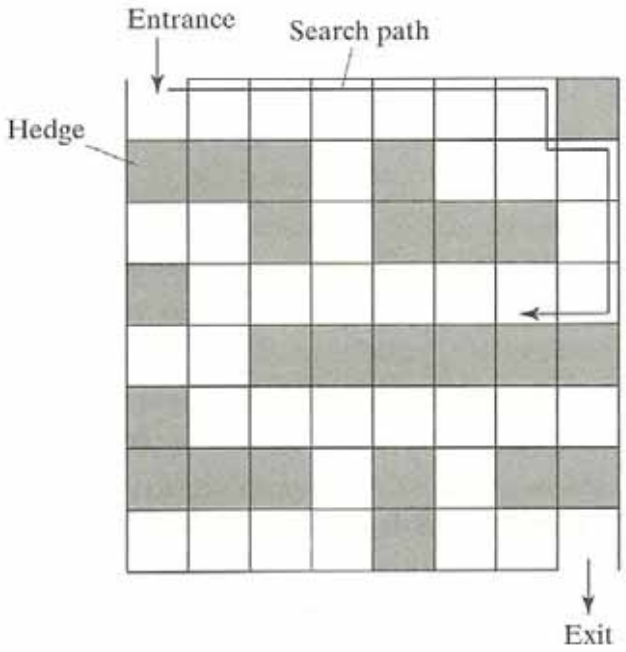


Figure 7.19 Sample maze for Problem 7-9.

7-10. A building has a number of interconnected rooms with a pot of gold in one, as illustrated in Figure 7.20. Draw a graph describing the plan of rooms where each vertex is a room. Doors connecting rooms are shown as edges between the rooms, as illustrated in Figure 7.21. Write a program that will find the path from the outside door to the chamber holding the gold. Notice that edges are bidirectional, and cycles may exist in the graph.

7-11. Historically, banks have used one or the other of two competing algorithms to handle the flow of customers at the teller stations within a bank: multiple-queue and single-queue. In the multiple-queue approach, each teller has a separate queue, similar to the lines at supermarkets.

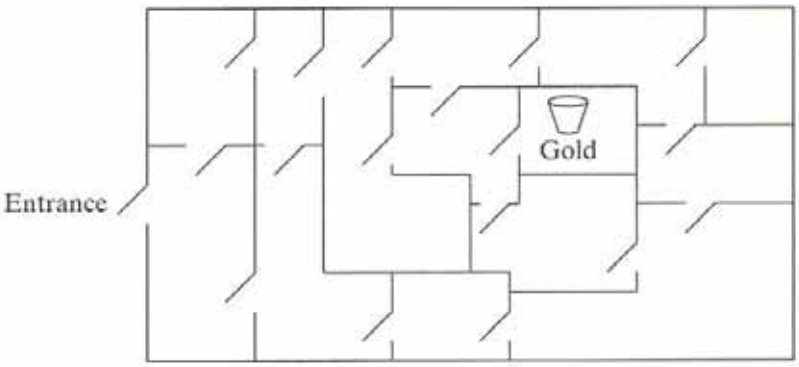


Figure 7.20 Plan of rooms for Problem 7-10.

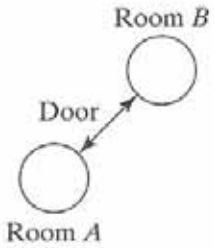


Figure 7.21 Graph representation for Problem 7-10.

In the purest form of this model, customers enter the bank, choose a queue to enter, and remain in it until served by the teller. One variation that is popular is to permit "queue hopping"; each person in each queue is constantly evaluating whether his or her chances of being served sooner would be enhanced by jumping to another queue. In the single-queue approach, there is only one queue.

The customer at the head of the queue is selected by the first teller completing a transaction. Your task is to simulate the pure multiple-queue and the single-queue approaches using a parallel program and prepare a one-page summary (management report) outlining the perceived advantages and disadvantages of each method given the following set of assumptions. In addition to such items as the average customer waiting times and maximum waiting times, gather whatever other statistics you feel are relevant in documenting your report's conclusions.

Assumptions:

1. There are five tellers.
2. All the queues are unlimited in size; customers will snake around the parking lot if necessary. However, the queues are empty at the start of business each day, and although no new customers are allowed into a queue after closing time, those already in the queue are permitted to complete their transactions.
3. Customers arrive randomly at the bank. Due to the bank's location near a major university, customers tend to be concentrated around the end of class times: 10 new customers arrive per minute (on average) between 10 minutes before and 10 minutes after the hour. Two new customers arrive per minute (on average) at all other times. The actual arrivals are random and are distributed evenly in the range of one to 19 arrivals per minute near the top of the hour and zero to four arrivals per minute at other times.
4. Each transaction takes a random amount of time to complete. On average, transactions take five minutes but are evenly distributed in the range from one to nine minutes. Each customer is considered a single transaction.

5. Run the simulations between the 9:00 A.M. and 6:00 P.M. (bank opening and closing times) for 100 days to generate the data from which you will draw conclusions for your summary report.

7-12. You are the president of a very large corporation employing nearly a million people. Your firm's personnel department has cleverly organized all the employees in a tree-style organization chart in which every employee reports to a supervisor but no supervisor has more than eight or fewer than two employees reporting to him or her. While it may be irrelevant, assume that the average number of employees reporting to a supervisor is five. Thus the average depth of the tree structure is roughly nine. (A little under 1,000,000 lowest-level employees report to about 200,000 first-level supervisors, who report to about 40,000 second-level supervisors, who report to about 8,000 third-level supervisors, etc.)

You have just heard from the U.S. Attorney General that one of your employees was indicted for something that may or may not affect your firm. You did not get the employee's name. Your task is to search the organization for the employee by following the official organization-chart personnel hierarchy. You are to do a breadth-first search, starting with the employees you directly supervise, until you identify the individual indicted. Note: You may assume that any nonindicted employee will answer "Not me!", while the employee who was indicted will answer "Yep, the feds got me!"

7-13. A table defines a collection of streets in a section of a major city. Many of the streets are one-way. In addition, there are several tunnels and bridges that allow the driver to skip over or under cross streets. The streets are all numbered. Even numbers are oriented east-west, while odd ones are oriented north-south. Each row of the table has the form

- street number being described
- cross street
- cross street
- mode (one-way or bidirectional)

As an example, one row might look like 13, 4, 6, 1, indicating that it is describing street number 13 in the block where it spans between streets 4 and 6, and is one-way in the direction from street 4 to street 6. (If the line had been 13, 6, 4, 1, then the street would have been one-way between streets 6 and 4.) Another row might look like 13, 6, 22, 2, indicating that street 13 is a two-way street and either a tunnel or a bridge in the section where it links streets 6 and 22 (with no entrances or exits from/to other cross streets between 6 and 22). Complete one (or more) of the following:

1. Find the number of paths that a taxi could use to get from one intersection to another in the city without passing through any intersection more than once.
2. Find the shortest path (fewest blocks traveled) that a taxi could use to get from one intersection to another without passing through any intersection more than once. Note: The only intersections that are associated with bridges or tunnels are those at the two ends.
3. Find the longest path (most blocks traveled) that a taxi could use to get from one intersection to another without passing through any intersection more than once. Note: The only intersections that are associated with bridges or tunnels are those at the two ends.

7-14. A brilliant, yet color-blind, researcher in the biology department has been growing cultured specimens of a dreaded bacterium in Petri dishes. While the culture solution is an opaque white, the bacteria are a pastel pink under visible light. This has hindered her greatly in the daily task of estimating the bacteria growth because she cannot discern yellow/orange/red hues.

She has rigged a digitizing camera that feeds data directly into a computer, and has hired you to write a scanning program that will calculate the percentage of the surface of the Petri dish covered by the bacteria. In addition, your program is to display the surface of the Petri dish in hues of blue/green.

After some initial experimentation, you have determined that an area of the Petri dish, center coordinates (x, y), has an average hue in the range from white to pink that depends upon both the (x, y) coordinates and the length of time, t, that the experiment has been running. For reasons that are not entirely clear, the exact relationship seems to be

$$\frac{t}{100} + \frac{(x+y)}{x_{\max} + y_{\max}} = Z$$

where the hue throughout a region is white if $Z < 0.95$ and pink otherwise. Your program is to compute and display the bacterium distribution across the Petri dish at a particular experiment time, t. Implement it so that you may zoom in on any particular point. Note: This should be computationally similar to a time-varying fractal, although the picture will not be nearly as jagged.

7-15. Lately, the TV, newspapers, and movies had been filled with stories about aliens, or so it seemed to Tom. Thus, when he was approached by an odd-looking stranger who was posing a multidimensional recursion problem to the people who lived in his apartment complex, Tom simply took it in stride. While it was vaguely discomforting to know that his family might never see him again if he failed to solve the problem, he was confident enough in his math skills to put aside all worries.

The only concern Tom had about the problem was that the aliens seemed much more at ease in dealing with dimensions greater than 3 than he was. But Tom was confident in his abilities and immediately dug into it.

Given an N-dimensional sphere of radius r, centered at the origin of an N-dimensional coordinate system, compute the number of integer coordinate points inside the sphere. The following are examples provided by the aliens for checking work:

(i) A three-dimensional sphere of radius 1.5 has 19 integer coordinate points within the sphere:

five points in the circle formed when the first coordinate is −1:

(−1, 0, 0), (−1, 0, 1), (−1, 0, −1), (−1, −1, 0), (−1, 1, 0),

five more in the circle when the first coordinate is 1:

(1, 0, 0), (1, 0, 1), (1, 0, −1), (1, −1, 0), (1, 1, 0),

and nine points in the circle when the first coordinate is 0:

(0, 0, 0), (0, 1, 0), (0, 1, 1), (0, 1, −1), (0, −1, 0), (0, −1, 1), (0, −1, −1), (0, 0, −1), and (0, 0, 1).

(ii) A two-dimensional sphere of radius 2.05 has 13 integer coordinate points within the sphere:

(0, 0), (−1, 0), (−2, 0), (1, 0), (2, 0), (−1, −1), (−1, 1), (1, −1), (1, 1), (0, −2), (0 −1), (0, 1), (0, 2).

(iii) A one-dimensional sphere of radius 25.5 has 51 integer coordinate points:

(±25, ±24, ±23, … ±1, 0).