

## Part I

### Preliminaries

#### Chapter 1

#### Parallel Computing

- 1.1 Bigger Problems, Faster Answers . . . . . 4
- 1.2 Applications for Parallel Computing . . . . . 6
- 1.3 For Further Information . . . . . 11

#### Chapter 2

#### Parallel Computers

- 2.1 A Brief History of Parallel Computers . . . . . 16
- 2.2 CPU Hardware . . . . . 18
- 2.3 SMP Parallel Computers . . . . . 22
- 2.4 Cluster Parallel Computers . . . . . 22
- 2.5 Hybrid Parallel Computers . . . . . 25
- 2.6 Computing Grids . . . . . 26
- 2.7 GPU Coprocessors . . . . . 27
- 2.8 SMPs, Clusters, Hybrids: Pros and Cons . . . 28
- 2.9 Parallel Programming Libraries . . . . . 30
- 2.10 For Further Information . . . . . 31

#### Chapter 3

#### How to Write Parallel Programs

- 3.1 Patterns of Parallelism . . . . . 36
- 3.2 Result Parallelism . . . . . 36
- 3.3 Agenda Parallelism . . . . . 38
- 3.4 Specialist Parallelism . . . . . 41
- 3.5 Clumping, or Slicing . . . . . 43
- 3.6 Master-Worker . . . . . 44
- 3.7 For Further Information . . . . . 46

#### Chapter 4

#### A First Parallel Program

- 4.1 Sequential Program . . . . . 48
- 4.2 Running the Sequential Program . . . . . 51
- 4.3 SMP Parallel Program . . . . . 52
- 4.4 Running the Parallel Program . . . . . 56
- 4.5 Running on a Regular Computer . . . . . 57
- 4.6 The Rest of the Book . . . . . 58
- 4.7 For Further Information . . . . . 59

#### Part I Exercises . . . . . 61

## Part II

### SMPs

#### Chapter 5

#### Massively Parallel Problems

- 5.1 Breaking the Cipher . . . . . 68
- 5.2 Preparing the Input . . . . . 69
- 5.3 Sequential Key Search Program . . . . . 71
- 5.4 Transitioning to a Parallel Program . . . . . 75
- 5.5 For Further Information . . . . . 76

#### Chapter 6

#### SMP Parallel Programming

- 6.1 Parallel Team . . . . . 78
- 6.2 Parallel Region . . . . . 79
- 6.3 Parallel For Loop . . . . . 82
- 6.4 Variables . . . . . 84
- 6.5 For Further Information . . . . . 87

#### Chapter 7

#### Massively Parallel Problems, Part 2

- 7.1 AES Key Search Parallel Program Design . . . 90
- 7.2 AES Key Search Parallel Program Code . . . 92
- 7.3 Early Loop Exit . . . . . 96

#### Chapter 8

#### Measuring Speedup

- 8.1 Speedup Metrics . . . . . 100
- 8.2 Amdahl's Law . . . . . 101
- 8.3 Measuring Running Time . . . . . 105
- 8.4 FindKeySmp Running Time Measurements . . 107
- 8.5 For Further Information . . . . . 109

#### Chapter 9

#### Cache Interference

- 9.1 Origin of Cache Interference . . . . . 112
- 9.2 Eliminating Cache Interference . . . . . 114
- 9.3 FindKeySmp3 Measurements . . . . . 116
- 9.4 For Further Information . . . . . 119

#### Chapter 10

#### Measuring Sizeup

- 10.1 Sizeup Metrics . . . . . 122
- 10.2 Gustafson's Law . . . . . 122

10.3 The Problem Size Laws . . . . .	124	15.4 Reduction Operators . . . . .	207
10.4 Measuring Sizeup . . . . .	127	15.5 Parallel Version with Reduction . . . . .	209
10.5 FindKeySmp3 Sizeup Data . . . . .	128	15.6 Performance Comparison . . . . .	213
10.6 Speedup or Sizeup? . . . . .	130	15.7 Critical Sections . . . . .	215
10.7 For Further Information . . . . .	131	15.8 Parallel Version with Critical Section . . . . .	216
Chapter 11		15.9 Summary: Combining Partial Results . . . . .	220
<b>Parallel Image Generation</b>		Chapter 16	
11.1 The Mandelbrot Set . . . . .	136	<b>Sequential Dependencies</b>	
11.2 Color Images . . . . .	137	16.1 Floyd's Algorithm . . . . .	224
11.3 Sequential Program . . . . .	139	16.2 Input and Output Files . . . . .	226
11.4 Parallel Program . . . . .	145	16.3 Sequential Program . . . . .	229
11.5 For Further Information . . . . .	150	16.4 Parallelizing Floyd's Algorithm . . . . .	231
Chapter 12		16.5 Parallel Program with Row Slicing . . . . .	232
<b>Load Balancing</b>		16.6 Parallel Program with Column Slicing . . . . .	239
12.1 Load Balance . . . . .	154	16.7 For Further Information . . . . .	242
12.2 Achieving a Balanced Load . . . . .	157	Chapter 17	
12.3 Parallel For Loop Schedules . . . . .	158	<b>Barrier Actions</b>	
12.4 Parallel Program with Load Balancing . . . . .	161	17.1 One-Dimensional Continuous Cellular Automata . . . . .	246
12.5 For Further Information . . . . .	165	17.2 Rational Arithmetic . . . . .	249
Chapter 13		17.3 Improving Memory Scalability . . . . .	251
<b>Reduction</b>		17.4 Sequential Program . . . . .	252
13.1 Estimating pi Using Random Numbers . . . . .	168	17.5 Barrier Actions . . . . .	256
13.2 Sequential Program . . . . .	170	17.6 Parallel Program . . . . .	259
13.3 Parallel Program . . . . .	172	17.7 For Further Information . . . . .	264
13.4 The Reduction Pattern . . . . .	175	Chapter 18	
13.5 Parallel Program with Reduction . . . . .	177	<b>Overlapping</b>	
13.6 The Second Flaw . . . . .	180	18.1 Overlapped Computation and I/O . . . . .	268
13.7 For Further Information . . . . .	181	18.2 Parallel Sections . . . . .	271
Chapter 14		18.3 Nested Parallel Regions . . . . .	274
<b>Parallel Random Number Generation</b>		18.4 Parallel Program with Overlapping . . . . .	276
14.1 Parallel PRNG Patterns . . . . .	184	<b>Part II Exercises</b> . . . . .	285
14.2 Pseudorandom Number Generator Algorithms . . . . .	187	<b>Part III</b>	
14.3 A Parallel PRNG Class . . . . .	190	<b>Clusters</b>	
14.4 Parallel Program with Sequence Splitting . . . . .	191	Chapter 19	
14.5 Parting Remarks . . . . .	194	<b>A First Cluster Parallel Program</b>	
14.6 For Further Information . . . . .	195	19.1 Sequential Program . . . . .	300
Chapter 15		19.2 Running the Sequential Program . . . . .	301
<b>Reduction, Part 2</b>		19.3 Cluster Parallel Program . . . . .	302
15.1 Histogram of the Mandelbrot Set . . . . .	198	19.4 Running the Parallel Program . . . . .	306
15.2 Sequential Version . . . . .	200		
15.3 Parallel Version without Reduction . . . . .	203		

## Chapter 20

**Parallel Message Passing**

- 20.1 Communicators . . . . .310
- 20.2 Point-to-Point Communication. . . . .311
- 20.3 Collective Communication. . . . .314

## Chapter 21

**Massively Parallel Problems, Part 3**

- 21.1 Cluster Parallel Program Design. . . . .330
- 21.2 Parallel Key Search Program . . . . .331
- 21.3 Parallel Program Speedup . . . . .334
- 21.4 Parallel Program Sizeup. . . . .336
- 21.5 Early Loop Exit . . . . .338

## Chapter 22

**Data Slicing**

- 22.1 Buffers . . . . .346
- 22.2 Single-Item Buffers . . . . .347
- 22.3 Array Buffers. . . . .348
- 22.4 Matrix Buffers. . . . .350
- 22.5 For Further Information. . . . .354

## Chapter 23

**Load Balancing, Part 2**

- 23.1 Collective Communication: Gather. . . . .356
- 23.2 Parallel Mandelbrot Set Program . . . . .358
- 23.3 Master-Worker . . . . .365
- 23.4 Master-Worker Mandelbrot Set Program. . . . .368

## Chapter 24

**Measuring Communication Overhead**

- 24.1 Measuring the Time to Send a Message. .382
- 24.2 Message Send-Time Model . . . . .388
- 24.3 Applying the Model. . . . .391
- 24.4 Design with Reduced Message Passing . .392
- 24.5 Program with Reduced Message Passing . .394
- 24.6 Message Scatter and Gather Time Models . . . . .402
- 24.7 Intra-Node Message Passing . . . . .403

## Chapter 25

**Broadcast**

- 25.1 Floyd's Algorithm on a Cluster. . . . .410
- 25.2 Collective Communication: Broadcast . .412
- 25.3 Parallel Floyd's Algorithm Program. . . .414
- 25.4 Message Broadcast Time Model. . . . .417

- 25.5 Computation Time Model . . . . .419

- 25.6 Parallel Floyd's Algorithm Performance. . .422

## Chapter 26

**Reduction, Part 3**

- 26.1 Estimating pi on a Cluster . . . . .428
- 26.2 Collective Communication: Reduction . .429
- 26.3 Parallel pi Program with Reduction . . . .432
- 26.4 Mandelbrot Set Histogram Program. . . .434

## Chapter 27

**All-Gather**

- 27.1 Antiproton Motion . . . . .446
- 27.2 Sequential Antiproton Program . . . . .455
- 27.3 Collective Communication: All-Gather . .460
- 27.4 Parallel Antiproton Program. . . . .464
- 27.5 Computation Time Model . . . . .470
- 27.6 Parallel Program Performance . . . . .473
- 27.7 The Gravitational  $N$ -Body Problem. . . .477
- 27.8 For Further Information. . . . .478

## Chapter 28

**Scalability and Pipelining**

- 28.1 Scalability. . . . .482
- 28.2 Pipelined Message Passing . . . . .484
- 28.3 Point-to-Point Communication: Send-Receive. . . . .487
- 28.4 Pipelined Antiproton Program . . . . .488
- 28.5 Pipelined Program Performance. . . . .496

## Chapter 29

**Overlapping, Part 2**

- 29.1 Overlapped Computation and Communication . . . . .500
- 29.2 Non-Blocking Send-Receive. . . . .502
- 29.3 Pipelined Overlapped Antiproton Program . . . . .502
- 29.4 Computation Time Model . . . . .508
- 29.5 Pipelined Overlapped Program Performance . . . . .509

## Chapter 30

**All-Reduce**

- 30.1 A Heat Distribution Problem . . . . .514
- 30.2 Sequential Heat Distribution Program . .523
- 30.3 Collective Communication: All-Reduce . .529

30.4 Mesh Element Allocation and Communication . . . . .	530
30.5 Parallel Heat Distribution Program . . . . .	534
30.6 Parallel Program Performance . . . . .	544
30.7 For Further Information . . . . .	545

## Chapter 31

### All-to-All and Scan

31.1 The Kolmogorov-Smirnov Test . . . . .	548
31.2 Block Ciphers as PRNGs . . . . .	550
31.3 Sequential K-S Test Program . . . . .	551
31.4 Parallel K-S Test Design . . . . .	555
31.5 Parallel K-S Test Program . . . . .	558
31.6 Parallel K-S Test Program Performance . . . . .	564
31.7 Collective Communication: All-to-All and Scan . . . . .	565
31.8 For Further Information . . . . .	569

## Part III Exercises . . . . . 571

## Part IV

### Hybrid SMP Clusters

## Chapter 32

### Massively Parallel Problems, Part 4

32.1 Hybrid Parallel Program Design . . . . .	584
32.2 Parallel Key Search Program . . . . .	587
32.3 Parallel Program Performance . . . . .	590

## Chapter 33

### Load Balancing, Part 3

33.1 Load Balancing with One-Level Scheduling . . . . .	596
33.2 Hybrid Program with One-Level Scheduling . . . . .	598
33.3 Program Performance with One-Level Scheduling . . . . .	605
33.4 Load Balancing with Two-Level Scheduling . . . . .	607
33.5 Hybrid Program with Two-Level Scheduling . . . . .	608
33.6 Program Performance with Two-Level Scheduling . . . . .	615

## Chapter 34

### Partitioning and Broadcast, Part 2

34.1 Floyd's Algorithm on a Hybrid . . . . .	622
34.2 Hybrid Parallel Floyd's Algorithm Program . . . . .	623
34.3 Computation-Time Model . . . . .	628
34.4 Hybrid Floyd's Algorithm Performance . . . . .	628

## Chapter 35

### Parallel Data-Set Querying

35.1 Data Sets and Queries . . . . .	634
35.2 Parallel Data-Set Querying Strategies . . . . .	634
35.3 The Prime Counting Function . . . . .	636
35.4 Sieving . . . . .	636
35.5 Sequential Prime Counting Program . . . . .	644
35.6 Hybrid Parallel Prime Counting Program . . . . .	645
35.7 For Further Information . . . . .	654

## Part IV Exercises . . . . . 657

## Part V

### Applications

## Chapter 36

### MRI Spin Relaxometry

36.1 MRI Scanning . . . . .	666
36.2 Spin Relaxometry Analysis . . . . .	670
36.3 Sequential Spin Relaxometry Program . . . . .	675
36.4 Cluster Parallel Program Design . . . . .	686
36.5 Parallel Spin Relaxometry Program . . . . .	688
36.6 Parallel Program Performance . . . . .	697
36.7 Displaying the Results . . . . .	698
36.8 Acknowledgments . . . . .	700
36.9 For Further Information . . . . .	701

## Chapter 37

### Protein Sequence Querying

37.1 Protein Sequences . . . . .	704
37.2 Protein Sequence Alignment . . . . .	705
37.3 A Protein Sequence Query Example . . . . .	714
37.4 Sequential Program . . . . .	718
37.5 Parallel Program, Version 1 . . . . .	728
37.6 Parallel Program, Version 2 . . . . .	742
37.7 Parallel Program Performance . . . . .	746

37.8 Smith-Waterman vs. FASTA and BLAST . . .	749
37.9 For Further Information . . . . .	749

## Chapter 38

### Phylogenetic Tree Construction

38.1 Phylogeny . . . . .	754
38.2 Distances . . . . .	757
38.3 A Distance Method: UPGMA . . . . .	759
38.4 Maximum Parsimony Method . . . . .	763
38.5 Maximum Parsimony with Exhaustive Search . . . . .	766
38.6 Maximum Parsimony with Branch-and-Bound Search . . . . .	778
38.7 Parallel Branch-and-Bound Search . . . . .	787
38.8 Acknowledgments . . . . .	797
38.9 For Further Information . . . . .	798

## Appendices

### A

#### OpenMP

A.1 OpenMP Programming . . . . .	802
A.2 OpenMP Features . . . . .	803
A.3 OpenMP Performance . . . . .	809
A.4 For Further Information . . . . .	813

### B

#### Message Passing Interface (MPI)

B.1 MPI Programming . . . . .	818
B.2 MPI Features . . . . .	821

B.3 MPI Performance . . . . .	824
B.4 For Further Information . . . . .	828

### C

#### Numerical Methods

C.1 Log-Log Plots . . . . .	834
C.2 Power Functions on a Log-Log Plot . . . . .	835
C.3 Power Function Curve Fitting . . . . .	836
C.4 Linear Regression . . . . .	838
C.5 General Linear Least-Squares Curve Fitting . . . . .	839
C.6 Quadratic Equations . . . . .	841
C.7 Cubic Equations . . . . .	841
C.8 For Further Information . . . . .	843

### D

#### Atomic Compare-and-Set

D.1 Blocking Synchronization . . . . .	846
D.2 Atomic Compare-and-Set . . . . .	848
D.3 Shared Variable Updating with Atomic CAS . . . . .	849
D.4 Limitations, Caveats . . . . .	850
D.5 For Further Information . . . . .	851

# Preface

## 1 Scope

*Building Parallel Programs (BPP)* teaches the craft of designing and coding—building—parallel programs, namely programs that employ multiple processors running all at once to solve a computational problem in less time than on one processor (speedup), to solve a larger problem in the same time (sizeup), or both.

*BPP* covers techniques for parallel programming on both major categories of parallel computers, SMPs and clusters. A shared memory multiprocessor (SMP) parallel computer consists of several central processing units sharing a common main memory, all housed in a single box. The “dual-core” and “multicore” computers now available from most vendors are examples of SMP parallel computers. An SMP parallel computer is programmed using multiple threads running in one process and manipulating shared data in the main memory. A cluster parallel computer consists of several processors, each with its own (non-shared) memory, interconnected by a dedicated high speed network. A cluster parallel computer is programmed using multiple processes, one per processor; each process manipulates data in its own memory and exchanges messages with the other processes over the network. *BPP* also covers techniques for parallel programming on a hybrid SMP cluster parallel computer (a cluster each of whose members is an SMP computer), which requires both multithreading and message passing in the same program.

*BPP* relies heavily on studying actual, complete, working parallel programs to teach the craft of parallel programming. The programs in the book are written in Java and use a Java class library I developed, **Parallel Java** (<http://www.cs.rit.edu/~ark/pj.shtml>). The Parallel Java Library hides the low-level details of multithreading and message passing, allowing the programmer to write parallel programs using high-level constructs.

For each parallel program examined in *BPP*, the source code is included in the text, interspersed with explanatory narrative. Learning to program begins with studying programs; so resist the temptation to gloss over the source code, and give the same attention to the source code as to the rest of the text. The program source files are also included in the Parallel Java Library. By downloading the Library, the source files can be studied using a text editor or an integrated development environment.

A major emphasis in *BPP* is the use of performance metrics—running time, speedup, efficiency, sizeup—in the design of parallel programs. For each parallel program studied, the book reports the program’s running time measurements on a real parallel computer with different numbers of processors, the book explains how to analyze the running time data to derive performance metrics, and the book explains how the metrics provide insights that lead to improving the program’s design. Consequently, *BPP* covers such performance-related topics as cache interference, load balancing, and overlapping in addition to parallel program design and coding.

## 2 Rationale

Why write another book on parallel programming? Like many textbooks, *BPP* grew out of my dissatisfaction with the state of parallel computing in general and my dissatisfaction with existing parallel programming textbooks.

Some books emphasize the concepts of parallel computing or the theory of parallel algorithms, but say little about the practicalities of writing parallel programs. *BPP* is not a parallel algorithms text, and it is not primarily a parallel computing concepts text; rather, *BPP* is a parallel *programming* text. But to teach practical parallel programming requires using a specific programming language. I have chosen to use Java. Why write a book on parallel programming in Java?

Three trends are converging to move parallel computing out of its traditional niche of scientific computations programmed in Fortran or C. First, parallel computing is becoming of interest in other domains that need massive computational power, such as graphics, animation, data mining, and informatics; but applications in these domains tend to be written in newer languages like Java. Second, Java is becoming the principal programming language students learn, both in high school AP computer science curricula and in college curricula. Recognizing this trend, the ACM Java Task Force has recently released a collection of resources for teaching introductory programming in Java (<http://jtf.acm.org>). Third, even desktop and laptop personal computers are now using multicore CPU chips. In other words, today's desktop and laptop PCs are SMP parallel computers, and in the near future even “regular” applications like word processors and spreadsheets will need to use SMP parallel programming techniques to take full advantage of the PC's hardware. Thus, there is an increasing need for *all* computing students to learn *parallel* programming as well as regular programming; and because students are learning Java as their main language, there is an increasing need for students to learn parallel programming *in Java*. However, there are no textbooks specifically about parallel programming in Java. *BPP* aims to fill this gap.

In the scientific computing arena, parallel programs are generally written either for SMPs or for clusters. Reinforcing this dichotomy are separate standard libraries—OpenMP (<http://openmp.org/wp/>) for parallel programming on SMPs, MPI (<http://www.mpi-forum.org/>) for parallel programming on clusters. Because SMPs perform best on certain kinds of problems and clusters perform best on other kinds of problems, it is important to learn both SMP and cluster parallel programming techniques. Most books focus either on SMPs or on clusters; none cover both in depth. *BPP* covers both.

However, in my experience OpenMP and MPI are difficult to teach, both because they are large and intricate libraries and because they are designed for programming in non-object-oriented Fortran and C, not object-oriented Java with which most students are familiar. The Parallel Java Library provides the capabilities of OpenMP and MPI in a unified, object-oriented API. Using the Parallel Java Library, *BPP* teaches the same parallel programming concepts and patterns as OpenMP and MPI programs use, but in an object-oriented style using Java, which is easier for students to learn. OpenMP aficionados will recognize Parallel Java's thread teams, parallel regions, work-sharing parallel for loops, reduction variables, and other features. MPI devotees will recognize Parallel Java's communicators and its message passing operations—send, receive, broadcast, scatter, gather, reduce, and others. Having mastered the concepts and patterns, students can then more easily learn OpenMP itself or MPI itself should they ever need to write OpenMP or MPI programs. Appendix A and Appendix B provide brief introductions to OpenMP and MPI and pointers to further information about them.



As multicore machines become more common, hybrid parallel computers—clusters of multicore machines—will become more common as well. However, there are no standard parallel programming libraries that integrate multithreading and message passing capabilities together in a single library. Hybrid parallel programs that use both OpenMP and MPI are not guaranteed to work on every platform, because MPI implementations are not required to support multithreading. While hybrid parallel programs can be written using only the process-based message passing paradigm with MPI, sending messages between different processes' address spaces on the same SMP machine often yields poorer performance than simply sharing the same address space among several threads. A hybrid parallel program should use the shared memory paradigm for parallelism within each SMP machine and should use the message passing paradigm for parallelism between the cluster machines. I developed the Parallel Java Library especially for writing parallel programs of this kind.

Furthermore, not much has been published about hybrid parallel programming techniques, and to my knowledge there are no textbooks that cover these techniques in depth. *BPP* aims to fill this gap by including cutting-edge material on hybrid parallel programming. Even in “plain” clusters where each node has only one CPU, considerable synergy arises from combining message passing parallel programming with multithreaded parallel programming, and *BPP* covers these techniques as well.

Finally, many existing parallel programming textbooks give short shrift to real-world applications. Either the books only cover general parallel programming techniques and never mention specific real-world applications at all, or the books merely describe examples of real-world applications and never study the actual code for such applications. In contrast, *BPP* covers three real-world parallel program codes in depth, one computational medicine problem and two computational biology problems. These applications reinforce the general parallel programming techniques covered in the rest of the book and show how to deal with practical issues that arise when building real parallel programs.

### 3. Target Audience and Prerequisites

*BPP* is aimed at upper division undergraduate students and graduate students taking a first course in parallel computing. *BPP* is also suitable for professionals needing to learn parallel programming.

I assume you know how to write computer programs, in Java, as typically taught in the introductory computer science course sequence.

I assume you are familiar with computer organization concepts such as CPU, memory, and cache, as typically taught in a computer organization or computer architecture course. Chapter 2 reviews these concepts as applied to parallel computers.

I assume you are familiar with what a thread is and with the issues that arise in multithreaded programs, such as synchronization, atomic operations, and critical sections. This material is typically taught in an operating systems course. However, when writing Parallel Java programs, you never have to write an actual thread or use low-level constructs like semaphores. The Parallel Java Library does all that for you under the hood.

I assume you are familiar with computer networking and with the notion of sending data in messages between processors over a network, as typically taught in a data communications or computer networks course. I also assume you are familiar with object serialization in Java. Again, when writing Parallel Java programs, you never have to open a socket or send a datagram. The Parallel Java Library does it for you under the hood.

Finally, I assume you have a mathematical background at the level of first-year calculus. We will be doing some derivatives and logarithms as we model and analyze parallel program performance.



## 4. Organization

*BPP* is organized into five parts.

Part I covers preliminary material. Chapter 1 defines what parallel computing is and gives examples of problems that benefit from parallel computing. After a brief history of parallel computing, Chapter 2 covers the parallel computer hardware and software concepts needed to develop parallel programs effectively. Chapter 3 describes the general process of designing parallel programs, based on the design patterns of Carriero and Gelernter. Chapter 4 gives a gentle introduction to parallel programming with Parallel Java.

Part II is an in-depth study of parallel programming on SMP parallel computers. Chapters 5, 6, and 7 introduce massively parallel problems and the programming constructs used to solve them, notably parallel loops. Chapters 8, 9, and 10 focus on performance. Chapter 8 introduces parallel program performance metrics, specifically running time, speedup, efficiency, and experimentally determined sequential fraction. Anomalies in these metrics reveal the issue of cache interference, covered in Chapter 9 along with techniques for avoiding cache interference. Chapter 10 looks at two further performance metrics, sizeup and sizeup efficiency. Returning to parallel program design, Chapter 11 discusses the issues that arise when using a parallel program to generate images. Chapter 12 covers parallel problems that need load balancing to achieve good performance, along with parallel loop schedules for load balancing. Chapter 13 introduces the topic of parallel reduction using thread safe shared variables. After an interlude in Chapter 14 on how to generate random numbers in a parallel program, Chapter 15 continues the topic of parallel reduction using reduction operators and critical sections. Chapter 16 looks at problems with sequential dependencies and how to partition such problems among multiple threads. Chapter 17 covers the barrier action, a construct for interspersing sequential code within parallel code. Concluding the coverage of SMP parallel programming, Chapter 18 shows how to increase performance by overlapping computation with I/O.

Part III shifts the focus to parallel programming on cluster parallel computers. After a gentle introduction in Chapter 19 to cluster parallel programming with Parallel Java, Chapter 20 covers the fundamental concepts of message passing. Chapter 21 describes how to solve massively parallel problems that do not need to communicate any data. Turning to problems that require communicating large amounts of data, Chapter 22 shows how to slice data arrays and matrices into pieces and send the pieces in messages. Chapter 23 illustrates how to do load balancing in a cluster parallel program using the master-worker pattern, along with data slicing. Chapter 24 derives a mathematical model for a cluster parallel program's communication time. This is used in succeeding chapters to model the parallel program's running time, providing insight into the maximum performance the program can achieve. Chapter 24 also shows how to redesign a master-worker program to reduce the communication time. The next three chapters focus on three so-called "collective communication" message passing operations and examine programs that illustrate each operation: Chapter 25, broadcast; Chapter 26, reduce; Chapter 27, all-gather. Chapter 28 focuses on scalability, describing how to assess a parallel program's performance and memory requirements as it scales up to larger problem sizes. Chapter 28 also covers pipelining, a technique that allows problems with large memory requirements to scale up by adding processors to the cluster. Chapter 29 returns to the topic of overlapping, showing how to do overlapped computation and communication in a cluster parallel program. The next two chapters conclude Part III with three more collective communication operations: Chapter 30, all-reduce; Chapter 31, all-to-all and scan.

Part IV brings the shared memory and message passing paradigms together to write programs for hybrid parallel computers. Chapter 32 shows how to solve massively parallel problems with parallel

programs containing both multiple processes and multiple threads per process. Chapter 33 covers load balancing on a hybrid parallel computer; the two degrees of freedom—load balancing among the processes, and load balancing among the threads within each process—provide considerable flexibility in the program design. Chapter 34 addresses the issues that arise when collective communication operations, specifically broadcast, must be done in a multithreaded message passing program. Chapter 35 covers parallel data set querying techniques, which are employed in many widely-used high performance scientific programs.

Parts II, III, and IV illustrate the general parallel programming techniques by studying the design, source code, and performance of several Parallel Java programs. These include programs to carry out a known plaintext attack on the Advanced Encryption Standard (AES) block cipher; compute an image of the Mandelbrot Set; estimate the value of  $\pi$  using a Monte Carlo technique; compute a histogram of the Mandelbrot Set image's pixels; find all shortest paths in a graph using Floyd's Algorithm; compute the evolution of a continuous cellular automaton; calculate the motion of antiprotons swirling around in a particle trap; calculate the temperature at each point on a metal plate by numerical solution of a partial differential equation; perform a Kolmogorov-Smirnov test on a random number generator; and compute the prime counting function using the Sieve of Eratosthenes. Many of the programs appear in several versions demonstrating different parallel programming techniques. Some of the programs appear in all three versions—SMP, cluster, and hybrid—to highlight the differences between the three paradigms. To emphasize the utility of parallel computing in domains other than the traditional ones of computational science and engineering, some of the problems solved with parallel programs in *BPP* are from “non-scientific” areas, such as cryptography and mathematics. However, the problems in Parts II, III, and IV were chosen solely for pedagogical reasons—to illustrate parallel programming techniques—and are not necessarily problems with any great significance in the real world.

Having covered parallel programming techniques in general, Part V goes on to solve three real-world problems requiring massive computation. Chapter 36 gives a cluster parallel program for magnetic resonance image (MRI) spin relaxometry, a computational medicine problem, based on curve fitting via nonlinear least squares. Chapter 37 gives two hybrid parallel programs for protein sequence querying, a computational biology problem, based on the Smith-Waterman local alignment algorithm. Chapter 38 gives an SMP parallel program for maximum parsimony phylogenetic tree construction, another computational biology problem, based on a parallel branch-and-bound algorithm.

Although many of the programs in *BPP* employ numerical algorithms, *BPP* is not a numerical methods textbook or a scientific computing textbook. The numerical methods are explained at a level sufficient to understand the parallel programs, but space does not permit detailed descriptions of the mathematics behind the numerical methods. To find out more about the numerical methods, see the references in the “For Further Information” section at the end of each chapter.

The book concludes with four appendices. Appendix A gives a brief introduction to OpenMP, describing OpenMP's features and comparing and contrasting OpenMP with Parallel Java. Appendix B does the same for MPI. Appendix A and Appendix B also compare the performance of OpenMP and MPI parallel programs written in C with the performance of the same parallel programs written in Java using the Parallel Java Library, demonstrating that Java programs can run as fast as or faster than C programs. Appendix C describes the numerical methods used throughout the book to analyze the parallel programs' performance. Appendix D covers the atomic compare-and-set operation, which can be used to achieve high-performance thread synchronization.

Exercises are included at the end of Parts I, II, III, and IV. Some are pencil-and-paper problems, some involve writing short parallel programs, some require investigating a program's behavior, some require a bit of research. I find that the best exercises integrate concepts from multiple chapters; thus, I have put the exercises at the end of each part of the book rather than at the end of each chapter.

## 5. Teaching

*BPP* contains more than enough material for a one-semester course on parallel computing. The book can also be used in a two-semester course sequence on parallel computing. The first semester would cover general SMP and cluster parallel programming techniques using Parts I–III. The second semester would cover hybrid parallel programming using Part IV and the real-world parallel programming applications in Part V, supplemented by the research literature and the instructor's own examples.

In the Computer Science Department at the Rochester Institute of Technology, I have been teaching the two-quarter parallel computing course sequence since 2005 using draft versions of the material in *BPP*. The “Parallel Computing I” course covers Parts I–III, the “Parallel Computing II” course covers Parts IV–V. I assign my students two term programming projects each quarter. In Parallel Computing I, the first project is to write an SMP parallel program to solve a stated problem; the second project is to write a cluster parallel program to solve the same problem. The students write a sequential and a parallel version of each program in Java using the Parallel Java Library, and they measure their programs' performance running on SMP and cluster parallel computers. In Parallel Computing II I do the same, except the second project is to write a hybrid parallel program.

Of course, to do parallel programming projects you will need a parallel computer. Nowadays it's easy to set up a parallel computer. Simply get a multicore server, and you have an SMP parallel computer. Or get several workstations and connect them with a dedicated high speed network like a 1-Gbps Ethernet, and you have a cluster parallel computer. Or do both. An interesting configuration is a hybrid parallel computer with four quad-core nodes. This gives you four-way parallelism for SMP parallel programs running with four threads on one node, sixteen-way parallelism for cluster parallel programs running with four processes on each of the nodes, and sixteen-way parallelism for hybrid parallel programs running with one process and four threads on each of the nodes. Larger multicore nodes and larger clusters will let you scale up your programs even further.

You will also need the Parallel Java Library. Parallel Java is free, GNU GPL licensed software and is available for download from my web site (<http://www.cs.rit.edu/~ark/pj.shtml>). The Library includes the Parallel Java middleware itself as well as all the parallel programs in this textbook, with source code, class files, and full Javadoc documentation. Parallel Java is written in 100% Java and requires only the Java 2 Standard Edition JDK 1.5 or higher to run on any machine; I have tested it on Linux and Solaris. Parallel Java is designed to be easy to install and configure; complete instructions are included in the Javadoc.

I am happy to answer general questions about the Parallel Java Library, receive bug reports, and entertain requests for additional features. Please contact me by email at [ark@cs.rit.edu](mailto:ark@cs.rit.edu). I regret that I am unable to provide technical support, specific installation instructions for your system, or advice about configuring your parallel computer hardware.

## 6. Acknowledgments

I would like to thank my student Luke McOmber, who helped write the first version of the Parallel Java Library in 2005.

I would like to thank the students in my Parallel Computing I and Parallel Computing II classes, who have helped me debug and refine the Parallel Java Library and the material in *BPP*: Michael Adams, Justin Allgyer, Brian Alliet, Ravi Amin, Corey Andalora, Sheehan Anderson, Michael Attridge, Justin Bevan, Benjamin Bloom, Patrick Borden, John Brothers, Andrew Brown, Kai Burnett, Patrick Campagnano, Stephen Cavilia, Samuel Chase, Mahendra Chauhan, Pierre Coirier, Christopher Connett, Sean Connolly, Adam Cornwell, Eric Cranmer, Mason Cree, Daniel Desiderio, Derek Devine, Joseph Dowling, Guilhem Duché, Christopher Dunnigan, Timothy Ecklund, Andrew Elble, Peter Erickson, James Fifield, Jonathan Finamore, Gaurav Gargate, Dalia Ghoneim, Andres Gonzalez, Jason Greguske, Allison Griggs, Craig Hammell, Joshua Harlow, Charles Hart, Jacob Hays, Darren Headrick, Ryan Hsu, Leighton Ing, Nilesh Jain, Sean Jordan, Alban Jouvin, Glenn Katzen, Pares Khatri, Dayne Kilheffer, John Klepack, Venkata Krishna Sistla, Michael Kron, Rachel Laster, Andrew LeBlanc, Joshua Lewis, Nicholas Lucaroni, Jon Ludwig, John MacDonald, Sushil Magdum, Jiong Mai, Daniel McCabe, Austin McChord, Sean McDermott, Michael McGovern, Eric Miedlar, David Mollitor, Kyle Morse, Adam Nabinger, Andrew Nachbar, Anurag Naidu, Paul Nicolucci, Brian Oliver, Shane Osaki, Derek Osswald, Robert Paradise, Kunal Pathak, Bryce Petrini, James Phipps, Michael Pratt, Paul Prystaloski, Michael Pulcini, Andrew Rader, Jessica Reuter, Aaron Robinson, Seth Rogers, Francis Rosa, Christopher Rouland, William Rummler, Omonbek Salaev, Mark Sanders, Josef Schelch, Bhavna Sharma, Michael Singleton, Amanda Sitterly, David Skidmore, Lukasz Skrzypek, Benjamin Solwitz, Brent Strong, Daniel Surdyk, Michael Szebenyi, Sakshar Thakkar, Jonathan Walsh, Borshin Wang, Kevin Watters, Jason Watt, Andrew Weller, Trevor West, Melissa Wilbert, Junxia Xu, and Andrew Yohn.

I would like to acknowledge Nan Schaller, who pioneered the RIT Computer Science Department's parallel computing courses. Nan also secured an NSF grant to acquire our first two SMP parallel computers. I would not have been able to develop the Parallel Java Library and the material in *BPP* if Nan had not paved the way.

I would like to thank the RIT Computer Science Department chairs, first Walter Wolf, then Paul Tymann, who procured capital funds for additional parallel computers to support teaching our parallel computing courses. As I write we have two 4-processor SMP parallel computers, two 8-processor SMP parallel computers, a 32-processor cluster parallel computer, and a 40-processor hybrid parallel computer with ten quad-core nodes. All of these use commodity off-the-shelf hardware. Our department's teaching facilities truly are "embarrassingly parallel."

I would like to thank Paul Austin and the Xerox Corporation. Paul was instrumental in obtaining a Xerox University Affairs grant for me which provided part of the funding for our hybrid parallel computer. Without this computer I would not have been able to develop and test the material on hybrid parallel programming that appears in Part IV.

I would like to thank our department's system administrator, James "Linus" Craig, and our hardware technician, Mark Stamer, for setting up our parallel computers, keeping them running, and for being generally supportive as I experimented with parallel computing middleware.

I would like to thank all the professionals at Course Technologies, especially Amy Jollymore, Alyssa Pratt, and Jill Braiewa, for their enthusiasm and effort in publishing the book. Without them you would

not be holding *BPP* in your hands. I would also like to thank the reviewers who took the time to scrutinize the manuscript and offer many helpful suggestions: Paul Gray, University of Northern Iowa; David Hemmendinger, Union College; Lubomir Ivanov, Iona College; April Kontostathis, Ursinus College; Tom Murphy, Contra Costa College; George Rudolph, The Citadel; Jim Teresco, Mount Holyoke College; and George Thiruvathukal, Loyola University Chicago. *BPP* is a better book because of these folks' efforts.

Lastly, I would like to thank my wife Margaret and my daughters Karen and Laura. Many were the days and evenings I would disappear into my upstairs sanctum to write. Thank you for your love and support during this long project.

I dedicate this book to my father, Edmund Kaminsky. Dad, you've waited a long time for this. Thank you for all your encouragement.

Alan Kaminsky  
December 2008

*This page intentionally left blank*

PART


# I

## Preliminaries

Chapter 1	
<b>Parallel Computing</b> .....	3
Chapter 2	
<b>Parallel Computers</b> .....	15
Chapter 3	
<b>How to Write Parallel Programs</b> .....	35
Chapter 4	
<b>A First Parallel Program</b> .....	47
<b>Part I Exercises</b> .....	61



*This page intentionally left blank*



CHAPTER

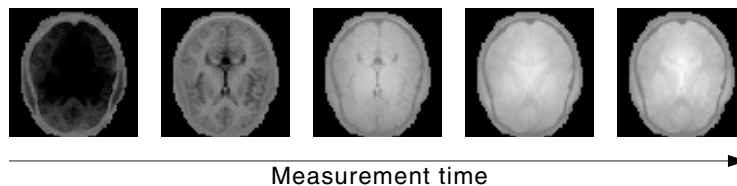
# 1

## Parallel Computing

in which we discover what parallel computing is; we learn how it can help us solve computational problems; and we survey several problems that benefit from parallel computing

## 1.1 Bigger Problems, Faster Answers

Many of today's computer applications require massive amounts of computation. Here's an example of a computational medicine problem involving **magnetic resonance imaging (MRI)**. MRI is a technique for making images of the inside of an organism, such as a living person's brain, without cutting the patient open. The MRI scanner sends a brief, high-intensity radio frequency pulse through the patient. The pulse reverses the orientation of the spins of the atoms in the patient. The MRI scanner then measures the atomic spins in a two-dimensional plane, or slice, through the subject as the atoms "relax," or return to their normal spin orientations. Each measurement takes the form of an  $N \times N$ -pixel image. The MRI scanner takes a sequence of these image snapshots at closely spaced time intervals (Figure 1.1).



**Figure 1.1** Sequence of magnetic resonance images for one slice of a brain

The rates at which the atoms' spins relax helps a doctor diagnose disease. In healthy tissue, the spins relax at certain rates. In diseased tissue, if abnormal chemicals are present, the spins relax at different rates. The sequence of measured spin directions and intensities for a given pixel can be analyzed to determine the spin relaxation rates in the tissue sample corresponding to that pixel. Such a **spin relaxometry analysis** requires sophisticated and time-consuming calculations on each pixel's data sequence to recover the underlying spin relaxation rates from the typically imperfect and noisy images. (In Chapter 36, we will examine MRI spin relaxometry analysis in more detail.)

One computer program that did the spin relaxometry analysis took about 76 seconds to do the calculations for a single pixel. To analyze all the pixels in, say, a  $64 \times 64$ -pixel image, a total of 4,096 pixels, would take about 311,000 seconds—over 3.5 days—of computer time. And this is for just one slice through the subject. A complete MRI scan involves *many* slices to generate a three-dimensional picture of the subject's interior. Clearly, the calculations need to be completed in a drastically shorter time for spin relaxometry analysis to be a useful diagnostic technique.

One alternative for reducing the calculation time is to get a faster computer. The earlier-noted timing measurement was made on a computer that was several years old. Running the same program on an

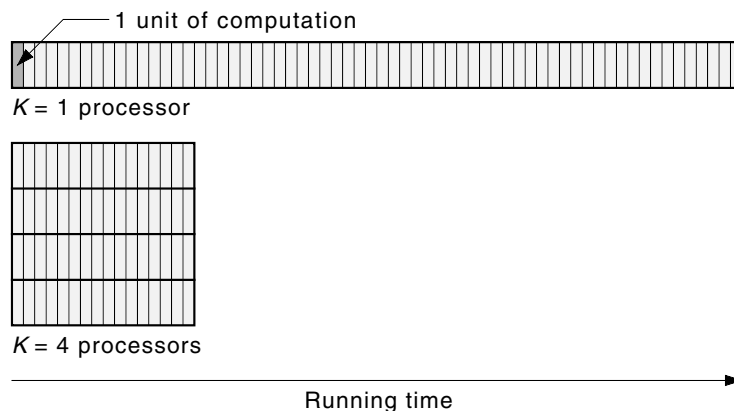
up-to-date computer that is 5 times faster than the original computer would take about 62,000 seconds to analyze a  $64 \times 64$ -pixel image, or about 17 hours instead of 3.5 days.

It used to be that computer clock speeds doubled roughly every two years. However, that trend finally may be ending. By 2004, CPU chips had achieved clock speeds in the 3 GHz range. If the trend had continued, clock speeds should have reached 12 GHz by 2008—but they did not. Instead, in late 2004, chip makers started moving away from the strategy of boosting chip performance by increasing the raw clock speed, opting instead to introduce architectural features such as “hyperthreaded” and “multi-core” chips (we will say more about these later). Although clock speeds do continue to increase, in the future there is little hope left for dramatically reduced calculation times from faster chips.

Another alternative for reducing the calculation time is to switch to a faster algorithm. Suppose we could devise a different program that was 100 times faster than the original; then running on the faster computer, it would take about 620 seconds—about 10 minutes—to analyze a  $64 \times 64$ -pixel image. Putting it another way, the calculation time per pixel would be 0.15 seconds instead of 76 seconds. However, algorithmic improvements can take us only so far. There comes a point where the fastest-known algorithm on the fastest available computer is still simply not fast enough.

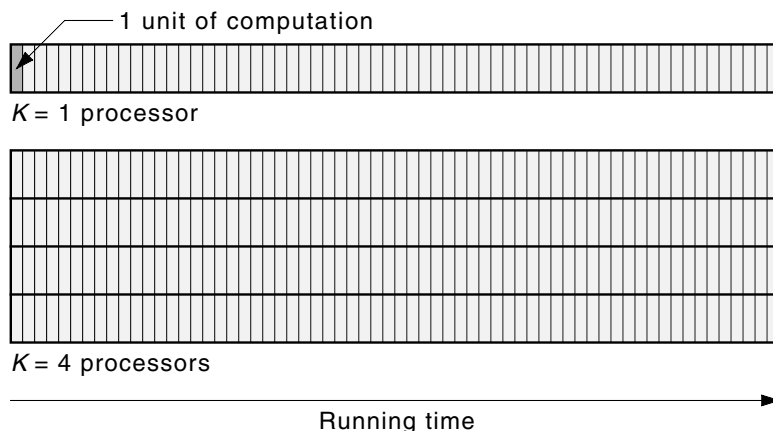
A third alternative is *to have several computers working on the problem simultaneously*. Say we have  $K$  computers instead of just one. We divide the problem up into  $K$  pieces and assign one piece to each computer. For the MRI spin relaxometry analysis problem, each computer analyzes  $(N \times N)/K$  pixels. Then all the computers go to work on their respective pieces at the same time. We say that the computers are executing **in parallel**, and we refer to the whole conglomeration as a **parallel computer**. Henceforth we will refer to the individual units within the parallel computer as **processors** to distinguish them from the parallel computer as a whole.

We could apply a parallel computer to the MRI spin relaxometry analysis problem in either of two ways. Suppose we have a 16-processor parallel computer. We divide the 4,096 pixels among the 16 processors. Because each processor has to do the calculations for only 256 pixels, and because all the processors are running at once, the computation takes only  $256 \times 0.15$  seconds, or about 39 seconds, instead of 10 minutes. The parallel computer let us *reduce the running time* by a factor of 16 while holding the problem size constant. Used in this way, a  $K$ -processor parallel computer ideally gives a **speedup** of  $K$  (Figure 1.2).



**Figure 1.2** Speedup with a parallel computer—same problem size,  $(1/K) \times$  running time

On the other hand, suppose we use our 16-processor parallel computer to analyze a magnetic resonance image with 16 times as many pixels—a 256×256-pixel image, which is actually the typical size of a magnetic resonance image used for a medical diagnosis. Either the image encompasses a larger area, or the image covers the same area at a finer resolution. Then the computation still takes the same 10 minutes, but we have analyzed a larger image. The parallel computer has let us *increase the problem size* by a factor of 16 while holding the running time constant. Used in this way, a  $K$ -processor parallel computer ideally gives a size increase, or **sizeup**, of  $K$  (Figure 1.3).



**Figure 1.3** Sizeup with a parallel computer— $K \times$  problem size, same running time

Of course, we can employ both strategies. A 64-processor parallel computer, for example, would let us analyze a 256×256-pixel image in one-fourth the time it takes a single computer to analyze a 64×64-pixel image. The more processors we add, the bigger the images we can analyze, and the faster we can get the answers.

To sum up, **parallel computing** is the discipline of employing multiple processors running all at once to solve the same problem in less time (speedup), to solve a larger problem in the same time (sizeup), or both. Another term often used is **high-performance computing (HPC)**, which emphasizes the improved performance parallel computing provides in solving larger problems or solving problems faster.

## 1.2 Applications for Parallel Computing

In 2004, the U.S. Office of Science and Technology Policy released a report titled “Federal Plan for High-End Computing.” This report lists four broad application areas—climate and weather, nanoscale science and technology, life sciences, and aerospace vehicle design—with problems requiring massive amounts of computation, that can and do benefit from parallel computing. Problems in other areas, such as astrophysics, mathematics, games, and animation, are also attacked using parallel computing. Here are a few examples of such problems:

**Weather forecasting.** In August 2005, Hurricane Katrina devastated the U.S. Gulf Coast, flooding the city of New Orleans, killing more than 1,800 people, and causing \$100 billion in damage. Computer models of the atmosphere, such as the Weather Research and Forecasting (WRF) Model, can predict

a storm's track (the path it will follow, where it will hit land) and intensity (wind speed). The WRF program takes a portion of the earth's atmosphere—a given region of the earth's surface, up to a given height above the surface—and divides this three-dimensional region into many small 3-D cells. The WRF program then uses physical models to calculate atmospheric conditions in each cell, as influenced by the neighboring cells, and advances the forecast in a series of many small time steps. The WRF program uses parallel computing to handle the calculations for large numbers of cells and time steps.

Accurate hurricane track and intensity forecasts can help officials decide how to prepare for a storm and where to evacuate if necessary. However, current hurricane forecasting programs are not all that accurate. With current models, track forecast errors can be as large as 70 kilometers (km), and wind speed forecast errors can be as large as 37 kilometers per hour (kph). A wind-speed shift of 37 kph can change a mere tropical storm to a Category 3 major hurricane. A track error of 70 km could cause officials to evacuate Boca Raton, Florida when they should have evacuated Miami.

To get more accurate predictions, the cells and the time steps in the model must be made smaller; this means that the model must include *more* cells and *more* time steps to cover the same geographic region and the same time span. For example, if the cell's dimensions are decreased by a factor of 2, the number of cells must increase by a factor of 8 to cover the same 3-D region. If the time step size is also decreased by a factor of 2, the number of time steps must increase by a factor of 2 to cover the same time span. Thus, the total amount of computation goes up by a factor of 16. This in turn means that even more powerful parallel computers and parallel programs will be needed to calculate the models.

**Climate modeling.** On what date will the rainy season begin in Brazil this year, so farmers will know when to plant their crops? Why is rainfall decreasing in the Indian subcontinent—could it be caused by pollution from burning wood for fuel and cooking? What effect will increased levels of atmospheric carbon dioxide have on the world's climate—none at all, or drastic warming that will melt the polar ice caps and inundate coastal cities? Computer-based climate models can answer these questions (and fan the controversies). The Community Climate System Model (CCSM), for example, models the atmosphere, ocean, sea ice, and land surface using a three-dimensional grid of cells like the WRF model. The CCSM program runs on a parallel computer to simulate the earth's climate over the entire globe for time spans of thousands of years. Because there is no end to the number of climatological features and the level of detail that can be included in climate simulation programs, such programs will continue to need the power of parallel computers well into the future.

**Protein sequence matching.** Imagine you are a biochemist. You have isolated a new protein from the creature you are studying, but you have no idea what the protein does—could it be an anti-cancer agent? Or is it just a digestive enzyme?

One way to get a clue to the protein's function is to match your protein against other proteins; if your protein closely matches proteins of known function, chances are your protein does something similar to the matching proteins. Chemically, a protein is a group of amino acids linked together into a long string. Twenty different amino acids are found in proteins, so a protein can be represented as a string of letters from a 20-character alphabet; this string is called the protein's "sequence." Protein sequence databases collect information about proteins, including their sequences and functions. The Swiss-Prot database, for example, contains well over 385,000 protein sequences ranging in length from 2 to 35,000 amino acids, with a median length of around 300 amino acids. You can determine your new protein's sequence and match it against the protein sequences in the database. Doing so is more complicated than looking up a

credit card number in a financial database, however. Rather than finding a single, exact match, you are looking for multiple, inexact but close matches.

The Basic Local Alignment Search Tool (BLAST) program is at present the premier tool for solving the preceding protein sequence matching problem. The BLAST program combines a “local alignment” algorithm, which matches a piece of one protein sequence against a piece of another protein sequence, with a search of all protein sequences in the database. Because local alignment is a computationally intensive algorithm and because the databases are large, parallel versions of BLAST are used to speed up the searches. In Chapter 37, we will design a parallel program for protein sequence database searching.

**Quantum computer simulation.** A quantum computer exploits quantum mechanical effects to perform large amounts of computation with small amounts of hardware. A quantum computer register with  $n$  qubits (quantum bits) can hold  $2^n$  different states at the same time via “quantum superposition” of the individual qubits’ states. Performing one operation on the quantum register updates all  $2^n$  states simultaneously, making some hitherto intractable algorithms practical. For example, in 1994, Peter Shor of AT&T Bell Laboratories published a quantum algorithm that can factor large composite numbers efficiently. If we could do that, we could break the RSA public key cryptosystem, which is the basis for secure electronic commerce on the Internet. The potential for solving problems with polynomial time algorithms on a quantum computer—that would otherwise require exponential time algorithms on a classical computer—has sparked interest in quantum algorithms.

Although small, specialized quantum computers have been built, it will be quite some time before useful general-purpose quantum computers become available. Nonetheless, researchers are forging ahead with quantum algorithm development. Lacking actual quantum computers to test their algorithms, researchers turn to quantum computer simulators running on classical computers. The simulators must do massive amounts of computation to simulate the quantum computer’s exponentially large number of states, making quantum computer simulation an attractive area for (classical) parallel computing. Several parallel simulator programs for quantum computers have been published.

**Star cluster simulation.** Astrophysicists are interested in the evolution of star clusters and even entire galaxies. How does the shape of the star cluster or galaxy change over time as the stars move under the influence of their mutual gravitational attraction? Many galaxies, including our own Milky Way, are believed to have a supermassive black hole (SMBH) at the center. How does the SMBH move as the comparatively much-lighter stars orbit the galactic center? What happens when two galaxies collide? While it’s unlikely for individual stars in the galaxies to collide with each other, the galaxies as a whole might merge, or they might pass through each other but with altered shapes, or certain stars might be ejected to voyage alone through intergalactic space.

There are theories that purport to predict what will happen in these scenarios. But because of the long time scales involved, millions or billions of years, there has been no way to test these theories by observing actual star clusters or galaxies. So astrophysicists have turned to observing the evolution of star clusters or galaxies *simulated in the computer*. In recent years, “computational astrophysics” has revolutionized the field and revealed a host of new phenomena for theorists to puzzle over.

The most general and powerful methods for simulating stellar dynamics, the so-called “direct  $N$ -body methods,” require enormous amounts of computation. The simulation proceeds as a series of time steps. At each time step, the gravitational force on each star from all the other stars is calculated, each star’s position and velocity are advanced as determined by the force, and the cycle repeats. A system of  $N$  stars requires  $O(N^2)$  calculations to determine the forces. To simulate, say, one million stars requires



$10^{12}$  force calculations—*on each and every time step*; and one simulation may run for thousands or millions of time steps. In Chapter 27, we will examine an  $N$ -body problem in more detail.

To run their simulations, computational astrophysicists turn to special purpose hardware. One example is the GRAPE-6 processor, developed by Junichiro Makino and his colleagues at the University of Tokyo. (GRAPE stands for GRAVity piPE.) The GRAPE-6 processor is a parallel processor that does only gravitational force calculations, but does them much, much faster than even the speediest general-purpose computer. Multiple GRAPE-6 processors are then combined with multiple general-purpose host processors to form a massively parallel gravitational supercomputer. Examples of such supercomputers include the GRAPE-6 system at the University of Tokyo and the “gravitySimulator” system built by David Merritt and his colleagues at the Rochester Institute of Technology.

**Mersenne primes.** Mersenne numbers—named after French philosopher Marin Mersenne (1588–1648), who wrote about them—are numbers of the form  $2^n - 1$ . If a Mersenne number is also a prime number, it is called a Mersenne prime. The first few Mersenne primes are  $2^2 - 1$ ,  $2^3 - 1$ ,  $2^5 - 1$ ,  $2^7 - 1$ , and  $2^{13} - 1$ . (Most Mersenne numbers are not prime.) The largest known Mersenne prime is  $2^{43,112,609} - 1$ , a whopper of a number with nearly 13 million decimal digits, discovered in August 2008 by the Great Internet Mersenne Prime Search (GIMPS) project.

Starting in 1996, the GIMPS project has been testing Mersenne numbers to find ever-larger Mersenne primes. The GIMPS project uses a “virtual parallel computer” to test candidate Mersenne numbers for primality in parallel. The GIMPS parallel computer consists of PCs and workstations contributed by volunteers around the globe. Each volunteer downloads and installs the GIMPS client program on his or her computer. The client runs as a lowest-priority process, and thus uses CPU cycles only when the computer is otherwise idle. The client contacts the GIMPS server over the Internet, obtains a candidate Mersenne number to work on, subjects the candidate to an array of tests to determine whether the candidate is prime, and reports the result back to the server. Because the candidate Mersenne numbers are so large, the primality tests can take days to weeks of computation on a typical PC. Since commencing operation, the GIMPS project has found 12 previously unknown Mersenne primes with exponents ranging from 1,398,269 to the aforementioned 43,112,609.

While Mersenne primes have little usefulness beyond pure mathematics, there is a practical incentive for continuing the search. The Electronic Frontier Foundation (EFF) has announced a \$100,000 prize for the first discovery of a ten-million-digit prime number—a prize now claimed by the GIMPS project. The EFF has also announced further prizes of \$150,000 for the first 100-million-digit prime number and \$250,000 for the first one-billion-digit prime number. While any prime number (not necessarily a Mersenne prime) qualifies for the prizes, the GIMPS project has perhaps the best chance at reaching these goals as well. According to their Web site, with these prizes, “EFF hopes to spur the technology of cooperative networking and encourage Internet users worldwide to join together in solving scientific problems involving massive computation.”

**Search for extraterrestrial intelligence (SETI).** Since 1997, researchers at the University of California, Berkeley have been using the radio telescope at Arecibo, Puerto Rico to search for signs of extraterrestrial intelligence. As the telescope scans the sky, the researchers record radio signals centered around a frequency of 1.42 GHz. (Because hydrogen atoms throughout the universe emit energy at this frequency, a fact of which any advanced civilization ought to be aware, SETI researchers feel that extraterrestrials who want to announce their presence would broadcast signals near this frequency.) These recorded signals are then analyzed to determine if they contain any “narrowband” signals—signals

confined to a small frequency range. Whereas most of the energy in the signal is “broadband” background noise spread out over a wide frequency range, a narrowband signal—like an AM radio, FM radio, television, or satellite signal—is more likely to have been generated by an intelligent being. Any detected narrowband signals are subjected to further scrutiny to eliminate signals of terrestrial origin. Signals that cannot be identified as terrestrial might just be extraterrestrial.

Because of the enormous amounts of radio signal data collected and the extensive computations needed to analyze the data to detect narrowband signals, the Berkeley researchers realized they needed a massively parallel computer. Rather than buy their own parallel supercomputer, they used the same approach as the GIMPS project and created the SETI@home project in 1999. Volunteers install the SETI@home client program on their computers. Running as a screen saver or as a low-priority background process, each client program contacts the SETI@home server over the Internet, downloads a “workunit” of radio signal data, analyzes the workunit, and sends the results back to the server. The SETI@home virtual supercomputer has analyzed nearly 300 million workunits so far, each workunit occupying 350 kilobytes of data and requiring 10 to 12 hours of computation on a typical PC, and has detected over 1.1 billion narrowband signals.

The SETI@home approach to parallel computing was so successful that the Berkeley researchers developed the Berkeley Open Infrastructure for Network Computing (BOINC), a general framework for Internet-based client-server parallel computation. Volunteers can donate compute cycles to any project that uses the BOINC infrastructure. Some 50 projects now use BOINC; they range from SETI@home itself to protein structure prediction to climate modeling.

Has SETI@home found any signs of extraterrestrial intelligence? The researchers are still working their way through the 1.1 billion narrowband signals that have been detected. So far, none can be conclusively stated as being of extraterrestrial origin.

**Chess.** On May 11, 1997, Garry Kasparov, chess grandmaster and then world chess champion, sat down opposite IBM chess computer Deep Blue for a six-game exhibition match. Kasparov and Deep Blue had met 15 months earlier, on February 10, 1996, and at that time Kasparov prevailed with three wins, two draws, and one loss. After extensive upgrades, Deep Blue was ready for a rematch. This time, the results were two wins for Deep Blue, one win for Kasparov, and three draws, for an overall score of Deep Blue 3.5, Kasparov 2.5. It was the first time a computer had won a match against a reigning human world chess champion. After his defeat, Kasparov demanded a rematch, but IBM refused and retired the machine. Deep Blue’s two equipment racks are now gathering dust at the National Museum of American History and the Computer History Museum.

Deep Blue was a massively parallel, special-purpose supercomputer, consisting of 30 IBM RS/6000 SP server nodes augmented with 480 specialized VLSI chess chips. It chose chess moves by brute force, analyzing plays at a rate of 200 million chess positions per second. It also had an extensive database of opening positions and endgames.

While Deep Blue is gone, computer chess research continues. The focus has shifted away from specialized parallel hardware to software programs running on commodity parallel machines. In November 2006, the Deep Fritz chess program, a parallel program running on a PC with two Intel dual-core CPU chips, beat then world chess champion Vladimir Kramnik with two wins and four draws. Many people now believe that in the realm of chess, human dominance over computers is at its end. Of course, it’s not really man versus machine, it’s human chess players against human computer builders and programmers.

**Animated films.** In 1937, Walt Disney made motion picture history with the animated feature film *Snow White and the Seven Dwarfs*. While Disney had been making animated short films since the 1920s, *Snow White* was the world's first *feature-length* animated film. In those days each frame of the film was laboriously drawn and colored by hand on celluloid. All that would change in 1995, when Pixar Animation Studios and Walt Disney Pictures released *Toy Story*, the world's first feature-length *computer-animated* film. Six years later, in 2001, DreamWorks Animation SKG debuted the computer-animated film *Shrek*, the first animated film to win an Academy Award. Since then, scarcely a year has gone by without several new feature-length computer-animated film releases.

During the early stages of production on a computer-animated film, the artists and designers work mostly with individual high-end graphics workstations. But when the time comes to “render” each frame of the final film, adding realistic surface textures, skin tones, hair, fur, lighting, and so on, the computation shifts to the “render farm”—a parallel computer with typically several thousand nodes, each node a multicore server. For *Toy Story*, the render farm had to compute a 77-minute film, with 24 frames per second and 1,536×922 pixels per frame—more than 157 billion pixels altogether. Despite the render farm's enormous computational power, it still takes hours or even days to render a single frame. As movie audiences come to expect ever-more-realistic computer-animated films, the render farms will continue to require increasingly larger parallel computers running increasingly sophisticated parallel rendering programs.

We've only scratched the surface, but perhaps you've gotten a sense of the broad range of problems that are being solved using parallel computing. The largest and most challenging of today's computer applications rely on parallel computing. It's an exciting area in which to work!

## 1.3 For Further Information

On the end of the trend towards ever-increasing CPU clock speeds:

- Herb Sutter. A fundamental turn toward concurrency in software. *Dr. Dobbs's Journal*, 30(3):16–22, March 2005.
- Craig Szydlowski. Multithreaded technology and multicore processors. *Dr. Dobbs's Journal*, 30(5):58–60, May 2005.

On applications for high performance computing:

- U.S. Office of Science and Technology Policy. Federal plan for high-end computing. May 10, 2004.  
[http://www.nitrd.gov/pubs/2004\\_hecrtf/20040702\\_hecrtf.pdf](http://www.nitrd.gov/pubs/2004_hecrtf/20040702_hecrtf.pdf)

On hurricane forecasting and the Weather Research and Forecast Model:

- Thomas Hayden. Super storms: No end in sight. *National Geographic*, 210(2):66–77, August 2006.

- J. Michalakes, J. Dudhia, D. Gill, T. Henderson, J. Klemp, W. Skamarock, and W. Wang. The Weather Research and Forecast Model: software architecture and performance. In *Proceedings of the 11th ECMWF Workshop on the Use of High Performance Computing In Meteorology*, 2004.  
[http://www.wrf-model.org/wrfadmin/docs/ecmwf\\_2004.pdf](http://www.wrf-model.org/wrfadmin/docs/ecmwf_2004.pdf)
- The Weather Research and Forecasting Model.  
<http://www.wrf-model.org/index.php>

On using parallel computing for weather modeling. “The Weather Man,” a fascinating science fiction story written in 1962 when computers were still young, computer networks not yet invented, and parallel computers barely beginning, nonetheless managed to give a remarkably prescient depiction of parallel computing on what are now known as networked workstation clusters:

- Theodore L. Thomas. “The Weather Man.” *Analog*, June 1962.

“The Weather Man” is reprinted in a couple more recent science fiction collections:

- Isaac Asimov and Martin H. Greenberg, editors. *Isaac Asimov Presents the Great SF Stories #24 (1962)*. DAW Books, 1992.
- David G. Hartwell and Kathryn Cramer, editors. *The Ascent of Wonder: The Evolution of Hard SF*. Tor Books, 1994.

On the Community Climate System Model:

- Special issue on the Community Climate System Model. *Journal of Climate*, 19(11), June 1, 2006.
- W. Collins, C. Bitz, M. Blackmon, G. Bonan, C. Bretherton, J. Carton, P. Chang, S. Doney, J. Hack, T. Henderson, J. Kiehl, W. Large, D. McKenna, B. Santer, and R. Smith. The Community Climate System Model Version 3 (CCSM3). *Journal of Climate*, 19(11):2122–2143, June 1, 2006.
- Community Climate System Model. <http://www.ccsm.ucar.edu/>

On protein sequence matching:

- The Universal Protein Resource (UniProt), including the Swiss-Prot protein sequence database. <http://www.uniprot.org/>
- National Center for Biotechnology Information. <http://www.ncbi.nlm.nih.gov/>
- S. Altschul, W. Gish, W. Miller, E. Myers, and D. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215(3):403–410, October 5, 1990.

On quantum computers, factoring algorithms, and simulators:

- D. DiVincenzo. Quantum computation. *Science*, 270(5234):255–261, October 13, 1995.
- C. Williams and S. Clearwater. *Ultimate Zero and One: Computing at the Quantum Frontier*. Copernicus, 2000.
- S. Loepp and W. Wootters. *Protecting Information: From Classical Error Correction to Quantum Cryptography*. Cambridge University Press, 2006.
- P. Shor. Algorithms for quantum computation: discrete logarithms and factoring. In *Proceedings of the 35th Annual Symposium on the Foundations of Computer Science*, 1994, pages 124–134.
- K. Obenland and A. Despain. A parallel quantum computer simulator. arXiv preprint arXiv:quant-ph/9804039v1, April 1998.  
<http://arxiv.org/abs/quant-ph/9804039v1>
- J. Niwa, K. Matsumoto, and H. Imai. General-purpose parallel simulator for quantum computing. arXiv preprint arXiv:quant-ph/0201042, January 2002.  
<http://arxiv.org/abs/quant-ph/0201042v1>
- K. De Raedt, K. Michielsen, H. De Raedt, B. Trieu, G. Arnold, M. Richter, T. Lippert, H. Watanabe, and N. Ito. Massive parallel quantum computer simulator. *Computer Physics Communications*, 176(2):121–136, January 15, 2007.

On gravitational supercomputers at the University of Tokyo and the Rochester Institute of Technology:

- J. Makino, T. Fukushige, M. Koga, and K. Namura. GRAPE-6: massively-parallel special-purpose computer for astrophysical particle simulations. *Publications of the Astronomical Society of Japan*, 55(6):1163–1187, December 2003.
- S. Harfst, A. Gualandris, D. Merritt, R. Spurzem, S. Portegies Zwart, and P. Berczik. Performance analysis of direct  $N$ -body algorithms on special-purpose supercomputers. *New Astronomy*, 12(5):357–377, July 2007.

On GIMPS and the EFF prime number prizes:

- The Great Internet Mersenne Prime Search. <http://www.mersenne.org/>
- Electronic Frontier Foundation Cooperative Computing Awards.  
[http://www.eff.org/awards/coop.php](http://www EFF.org/awards/coop.php)

On SETI@home:

- E. Korpela, D. Werthimer, D. Anderson, J. Cobb, and M. Lebofsky. SETI@home—massively distributed computing for SETI. *Computing in Science and Engineering*, 3(1):78–83, January 2001.
- D. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer. SETI@home: an experiment in public-resource computing. *Communications of the ACM*, 45(11):56–61, November 2002.
- SETI@home. <http://setiathome.berkeley.edu/>

On BOINC:

- D. Anderson. BOINC: a system for public-resource computing and storage. In *Proceedings of the Fifth IEEE/ACM International Workshop on Grid Computing*, 2004, pages 4–10.
- Berkeley Open Infrastructure for Network Computing. <http://boinc.berkeley.edu/>

On Deep Blue:

- F. Hsu, M. Campbell, and J. Hoane. Deep Blue system overview. In *Proceedings of the Ninth International Conference on Supercomputing (ICS'95)*, 1995, pages 240–244.
- F. Hsu. *Behind Deep Blue: Building the Computer that Defeated the World Chess Champion*. Princeton University Press, 2002.
- F. Hsu. Chess hardware in Deep Blue. *Computing in Science and Engineering*, 8(1):50–60, January 2006.

On rendering *Toy Story*:

- T. Porter and G. Susman. Creating lifelike characters in Pixar movies. *Communications of the ACM*, 43(1):25–29, January 2000.

On Pixar's and DreamWorks' render farms:

- M. Hurwitz. Incredible animation: Pixar's new technologies. November 2004. [http://www.uemedia.net/CPC/vfxpro/article\\_10806.shtml](http://www.uemedia.net/CPC/vfxpro/article_10806.shtml)
- S. Twombly. DreamWorks animation artists go over the top with HP technology. May 2006. [http://www.hp.com/hpinfo/newsroom/feature\\_stories/2006/06animation.html](http://www.hp.com/hpinfo/newsroom/feature_stories/2006/06animation.html)

The background of the top half of the page is an abstract image featuring a series of parallel, slightly curved lines that create a sense of depth and perspective, resembling a staircase or a modern architectural structure. A large, white, stylized number '2' is positioned on the right side of this section, partially overlapping the text 'CHAPTER'.

CHAPTER

# 2

## Parallel Computers

in which we briefly recount the history of parallel computers; we examine the characteristics of modern parallel computer hardware that influence parallel program design; and we introduce modern standard software libraries for parallel programming



## 2.1 A Brief History of Parallel Computers

To understand how modern parallel computers are built and programmed, we must take a quick look at the history of parallel computers.

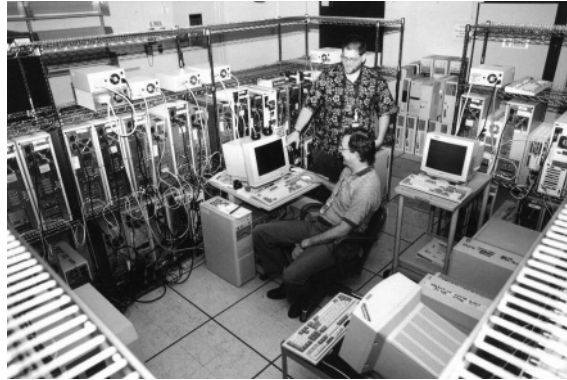
Up until the mid-1990s, there were no widely adopted standards in either parallel computer hardware or software. Many vendors sold parallel computers, but each vendor had its own proprietary designs for CPUs and CPU interconnection networks. Along with its proprietary hardware, each vendor provided its own proprietary languages and tools for writing parallel programs on its hardware—sometimes a variant of a scientific programming language such as Fortran, sometimes another language.

Because the hardware and software were mostly vendor-specific, parallel programs of that era tended not to be portable. You settled on a hardware vendor, then you used the vendor's supported parallel programming language to write your parallel programs. If you wanted to change vendors, you were faced with the daunting prospect of rewriting and re-debugging all your programs using the new vendor's programming language.

As the twentieth century drew toward its close, four events took place that signaled the beginning of a paradigm shift for parallel computing. First, in the late 1970s, Robert Metcalfe and David Boggs invented the Ethernet local area network at the Xerox Palo Alto Research Center; in 1980, the 10-Mbps Ethernet standard was published by Digital Equipment Corporation, Intel, and Xerox; in 1983, a variation was standardized as IEEE Std 802.3. Second, in 1981, the Internet Protocol (IP) and the Transmission Control Protocol (TCP), developed by Vinton Cerf and Robert Kahn, were published as Internet standards—"Request For Comments" (RFC) 791 and RFC 793. Third, also in 1981, IBM started selling the IBM PC, whose open architecture became the de facto standard for personal computers. By 1983, for the first time in the history of computing, there was an open standard computer hardware platform (the PC), an open standard local area network (Ethernet) for interconnecting computers, and an open standard network protocol stack (TCP/IP) for exchanging data between computers. Fourth, in 1991, Linus Torvalds released the first version of the Linux operating system, a free, Unix-like operating system for the PC, that included an implementation of TCP/IP.

In 1995, Thomas Sterling, Donald Becker, and John Dorband at the NASA Goddard Space Flight Center, Daniel Savarese at the University of Maryland, and Udaya Ranawake and Charles Packer at the Hughes STX Corporation published a paper titled "Beowulf: a parallel workstation for scientific computation." In this paper, they described what is now called a cluster parallel computer, built from off-the-shelf PC boards, interconnected by an off-the-shelf Ethernet, using the Internet standard protocols for communication, and running the Linux operating system on each PC. The Beowulf cluster's performance was equal to that of existing proprietary parallel computers costing millions of dollars, but because it was built from off-the-shelf components, the Beowulf cluster cost only a fraction as much.

A year later, William Hargrove and Forrest Hoffman at Oak Ridge National Laboratory proved that a parallel computer could be built for zero dollars. Lacking a budget to buy a new parallel computer, they instead took obsolete PCs that were destined for the landfill, loaded them with the free Linux operating system, and hooked them up into a Beowulf cluster. Dubbing their creation the “Stone SouperComputer” (Figure 2.1), after the fable of the soldier who cooked up “stone soup” from a small stone along with a bit of this and a bit of that contributed by the villagers in the story, eventually Hargrove and Hoffman had a cluster with 191 nodes.



<http://www.extremelinux.info/stonesoup/photos/1999-05/image01.jpg>

**Figure 2.1** The Stone Souper Computer

Once the PC, Ethernet, and TCP/IP became standardized, prices were driven down by mass production of microprocessor, memory, and Ethernet chips and cutthroat competition in the PC market. Today, computers are commodities you can buy in a store just like you can buy toasters and televisions—a state of affairs undreamed of when proprietary designs held sway. To get a parallel computer with a given amount of computing power, it usually costs much less to buy a multicore PC server, or to buy a bunch of PCs and a 1-Gbps Ethernet switch, from the corner computer store than to buy a proprietary computer; and because Linux is free, it doesn’t cost anything to equip these PCs with an operating system. While proprietary parallel computer companies are still in business, their computers primarily occupy a niche at the ultra-high-performance end of the spectrum. The majority of parallel computing nowadays takes place on commodity hardware.

Since 1993, the TOP500 List (at <http://www.top500.org>) has been tracking the 500 fastest supercomputers in the world, as measured by a standard benchmark (the LINPACK linear algebra benchmark). In the June 1993 TOP500 List, only 104 of the top 500 supercomputers (21%) used commodity CPU chips—Intel i860s and Sun SuperSPARCs. The rest of the top 500 (79%) used proprietary CPUs. In contrast, the June 2008 TOP500 list shows 498 of the top 500 supercomputers using commodity CPU chips from (in alphabetical order) AMD, IBM, and Intel. By the way, in June 1993, 97 of the top 500 supercomputers were still single-CPU systems. In June 2008, *all* the top 500 supercomputers were parallel computers of one kind or another, with anywhere from 80 to over 200,000 processors.

In the transition to commodity hardware, parallel programming also shifted away from using proprietary programming languages to using standard, non-parallel programming languages, chiefly Fortran, C, and C++, coupled with standard parallel programming libraries. The Parallel Virtual Machine (PVM)

library for programming cluster computers was first released in 1989. The Message Passing Interface (MPI) standard, also for programming cluster computers, was first released in 1994. The OpenMP standard for multithreaded parallel programming was first released in 1997. Like Linux, free versions of PVM, MPI, and OpenMP are widely available. Because these are hardware-independent standards, parallel programs written on one machine can be easily ported to another machine. The majority of parallel programming nowadays is done using a standard language and library.

Because parallel computing with commodity hardware and software is now firmly established, this book will focus on building parallel programs for commodity parallel computers. To build good parallel programs requires understanding the characteristics of parallel computer hardware and software that influence parallel program design. Next, we'll look at these characteristics and at the prevalent parallel computer architectures.

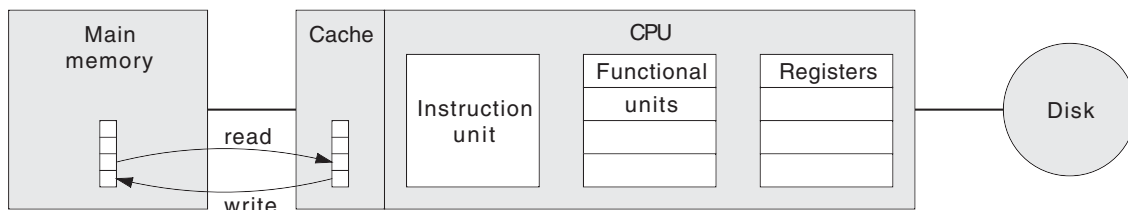
## 2.2 CPU Hardware

To help achieve the goal of ever-increasing computer performance (and sales), **central processing units (CPUs)** have become bewilderingly complex. A modern CPU may employ architectural features such as these:

- **Pipelined architecture.** While one instruction is being fetched from memory, another already-fetched instruction is being decoded, and several more already-decoded instructions are in various stages of execution. With multiple instructions in process at the same time in different stages of the pipeline, the CPU's effective computation speed increases.
- **Superscalar architecture.** The CPU has several functional units, each capable of executing a different class of instructions—an integer addition unit, an integer multiplication unit, a floating-point unit, and so on. If several instructions use different functional units and do not have other dependencies among each other, the CPU can execute all the instructions at once, increasing the CPU's effective speed.
- **Instruction reordering.** To utilize the CPU's pipeline and functional units to the fullest, the CPU may issue instructions in a different order from the order they are stored in memory, provided the results turn out the same.

High-level programming languages hide most of this architectural complexity from the programmer. The hardware itself, perhaps aided by the high-level language compiler or (in the case of Java) the virtual machine, takes care of utilizing the CPU's architectural features. A Java or C program, for example, does not have to be rewritten if it is going to be run on a superscalar CPU rather than a CPU with just one functional unit. However, there are two CPU architectural features that *do* make a difference in the way high-level language programs are written: cache memories, and symmetric multiprocessors.

**Cache memories.** As CPU speeds outpaced memory speeds, computer architects added a **cache memory** to avoid making a fast CPU wait for a slow main memory (Figure 2.2).



**Figure 2.2** Computer with cache memory

The main memory is large, typically gigabytes or hundreds of megabytes, but slow. The cache memory is much smaller than main memory, typically a few megabytes, but is also much faster than main memory. That is, it takes much less time for the CPU to read or write a word in cache memory than in main memory.

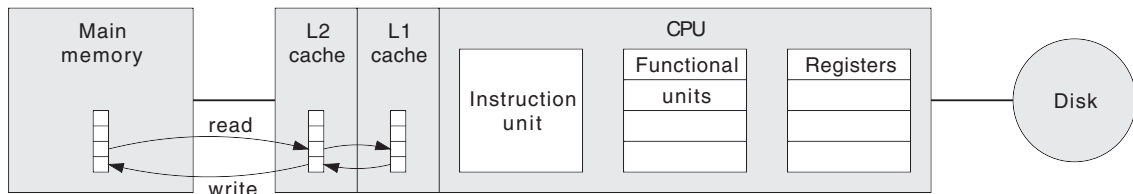
With the cache in place, when the CPU reads a word at a certain address, the CPU first checks whether the desired word has been loaded into the cache. If it has not—a **cache miss**—the CPU loads the entire **cache line** containing the desired word from main memory into the cache; then the CPU reads the desired word from the cache. The cache line size depends on the processor; 64 to 128 bytes is typical. Thereafter, when the CPU reads the same word, or reads another word located in the same cache line, there is a **cache hit**; the CPU reads the word directly from the cache and does not need to load it from main memory. Thus, if the **cache hit ratio**—the fraction of all memory accesses that are cache hits—is large, the CPU will read data words at nearly the speed of the fast cache memory and will seldom have to wait for the slow main memory.

As the CPU continues to load cache lines from main memory into the cache, eventually the cache becomes full. At the next cache miss, the CPU must replace one of the previously loaded cache lines with the new cache line. The cache’s **replacement policy** dictates which cache line to replace. Various replacement policies are possible, such as replacing a *randomly chosen* cache line, or replacing the cache line that has not been accessed for the longest amount of time (the *least recently used* cache line).

When the CPU writes a word at a certain address, the CPU must load the relevant cache line from main memory if there is a cache miss, then the CPU can replace the contents of the desired word in the cache with the new value. Subsequent reads of that word will retrieve the new value from the cache. However, after a write, the contents of the word in the cache do not match the contents of the word in main memory; the cache line is said to be **dirty**. The cache’s **write policy** dictates what to do about a dirty cache line. A *write-through cache* copies the dirty cache line to main memory immediately. A *write-back cache* copies the dirty cache line to main memory only when the cache line is being replaced.

The cache has a profound effect on program performance. A program’s **working set** consists of all the memory locations the program is currently accessing, both locations that contain instructions and locations that contain data. If the program’s working set fits entirely within the cache, the CPU will be able to execute the program as fast as it possibly can. This is often possible when the bulk of the program’s time is spent in a tight loop operating on a data structure smaller than the cache. To the extent that the program’s working set does not fit in the cache, the program’s performance will be reduced. In this case, the name of the game is to design the program to minimize the number of inevitable cache misses.

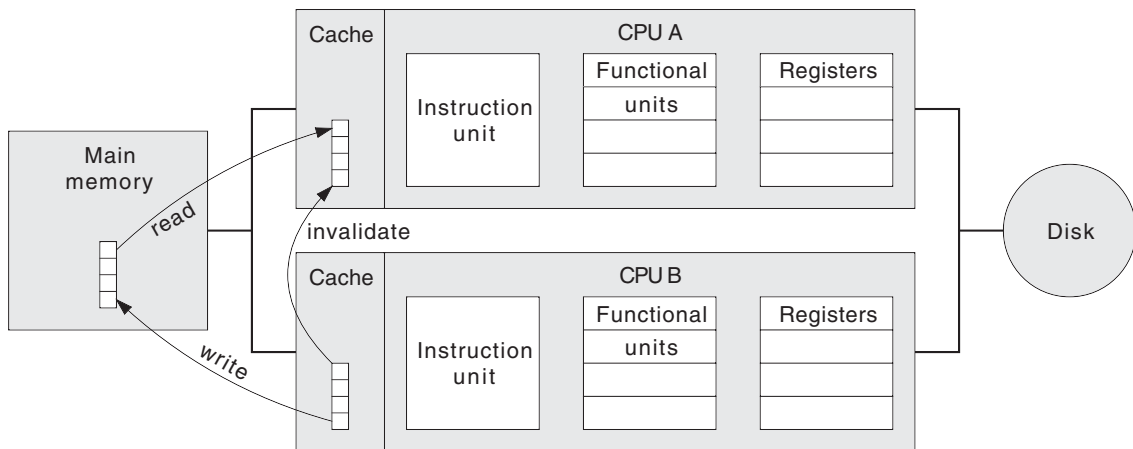
Some CPUs are so fast that even the cache is a performance bottleneck. Such CPUs use a **multilevel cache** (Figure 2.3).



**Figure 2.3** Computer with multilevel cache

A **level-1 (L1) cache** sits between the CPU and the **level-2 (L2) cache** (formerly the only cache). The L1 cache is even faster than, and smaller than, the L2 cache. The L1 cache bears the same relationship to the L2 cache as the L2 cache bears to main memory. From the programmer's point of view, whether the cache has multiple levels is less important than the cache's existence. The name of the game is still to design the program to minimize the number of cache misses.

**Symmetric multiprocessors.** To achieve performance gains beyond what is possible on a single CPU, computer architects replicated the CPU, resulting in a **symmetric multiprocessor** (Figure 2.4). It is called “symmetric” because all the processors are identical. Each processor is a complete CPU with its own instruction unit, functional units, registers, and cache. All the processors share the same main memory and peripherals. The computer achieves increased performance by running multiple threads of execution simultaneously, one on each processor.



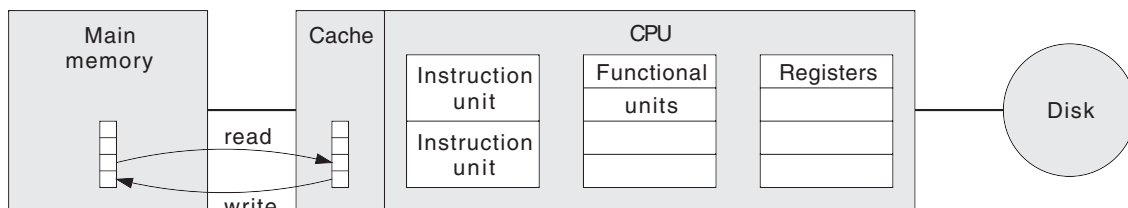
**Figure 2.4** Symmetric multiprocessor

Going to multiple processors, however, affects the caches' operation. Suppose CPU A reads the word at a certain address  $x$ , so that a copy of the relevant cache line is loaded from main memory into CPU A's cache. Suppose CPU B now writes a new value into the word at address  $x$ . CPU B's cache line is written back to main memory. However, CPU A's cache line no longer has the correct contents. The CPUs use a **cache coherence protocol** to bring the caches back to a consistent state. One popular cache coherence protocol uses **invalidation**. When CPU B writes its value into address  $x$ , CPU B sends an “invalidate” signal to tell CPU A that the contents of address  $x$  changed. In response, CPU A changes its cache state to

say that the cache line containing address  $x$  does not reside in the cache. This is called “invalidating” the cache line. The next time CPU A reads address  $x$ , CPU A will see that the cache line containing address  $x$  is not loaded, CPU A will reload the cache line from main memory, and CPU A will retrieve the value written by CPU B.

Note that writing a word in one CPU can slow down another CPU, because the other CPU has to re-read the word’s cache line from slow main memory. This **cache interference** effect can reduce a parallel program’s performance, and we will return to the topic of cache interference in Chapter 9.

Symmetric multiprocessor computers originally used a separate CPU chip for each processor (thread). However, as the number of transistors that could fit on a chip continued to increase, computer architects started to contemplate running more than one thread on a single chip. For example, instead of simultaneously issuing multiple instructions from a single instruction stream—a single thread—to multiple functional units, why not issue multiple instructions from *multiple* threads simultaneously to the functional units? Doing this for, say, two threads requires two instruction units, one to keep track of each thread’s instruction stream. The result is called a **hyperthreaded CPU** (Figure 2.5).



**Figure 2.5** Computer with hyperthreaded CPU

Two instruction units are by no means the limit. Some of today’s fastest supercomputers use **massively multithreaded processors (MMPs)** that can handle hundreds of simultaneous threads. As soon as one thread stalls, perhaps because it has to wait for data to be loaded from main memory into the cache, the CPU can instantly switch to another thread and keep computing.

As transistor densities continued to increase, it became possible to replicate the whole processor, not just the instruction unit, on a single chip. In other words, it became possible to put a symmetric multiprocessor on a chip. Such a chip is called a **multicore CPU**. Alternatively, the name may refer to the number of processors on the chip: a **dual-core CPU**; a **quad-core CPU**; and so on. Multicore chips can themselves be aggregated into symmetric multiprocessor systems, such as a four-processor computer comprising two dual-core CPU chips.

It used to be that you could increase an application program’s performance simply by trading in your old PC for a new one with a faster CPU chip. That won’t necessarily work any longer. Now that chips have become hyperthreaded or multicore, your new PC may very well have a multicore CPU with the same clock speed as, or even a slower clock speed than, your old PC. If your application is single-threaded, as many are, it can run only on one CPU of the multicore chip and thus may run *slower* on your new PC! Until applications are redesigned as multithreaded programs—*parallel* programs—users will not see much of a performance improvement when running applications on the latest multicore PCs.

Having examined the salient features of individual CPUs, we next look at different ways to combine multiple CPUs to make a complete parallel computer.

## 2.3 SMP Parallel Computers

There are three principal parallel computer architectures using commodity hardware: SMPs, clusters, and hybrids. There is also a variant called a computing grid. Parallel coprocessors using commodity graphics chips have also been introduced.

A **shared memory multiprocessor (SMP)** parallel computer is nothing more than a symmetric multiprocessor system (Figure 2.6). Each processor has its own CPU and cache. All the processors share the same main memory and peripherals.

A parallel program running on an SMP parallel computer (Figure 2.7) consists of one process with multiple threads, one thread executing on each processor. The process's program and data reside in the shared main memory. Because all threads are in the same process, all threads are part of the same **address space**, so all threads access the same program and data. Each thread performs its portion of the computation and stores its results in the shared data structures. If the threads need to communicate or coordinate with each other, they do so by reading and writing values in the shared data structures.

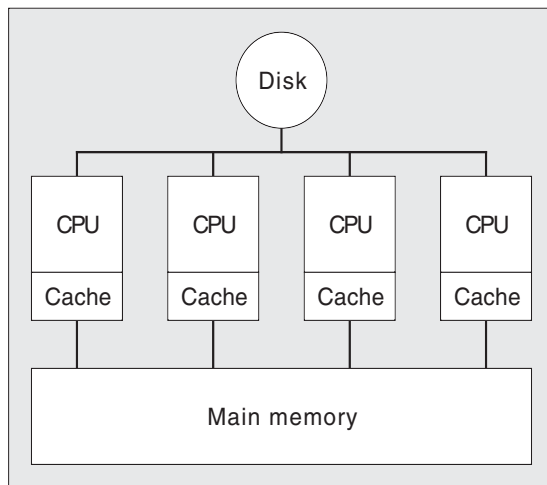


Figure 2.6 SMP parallel computer

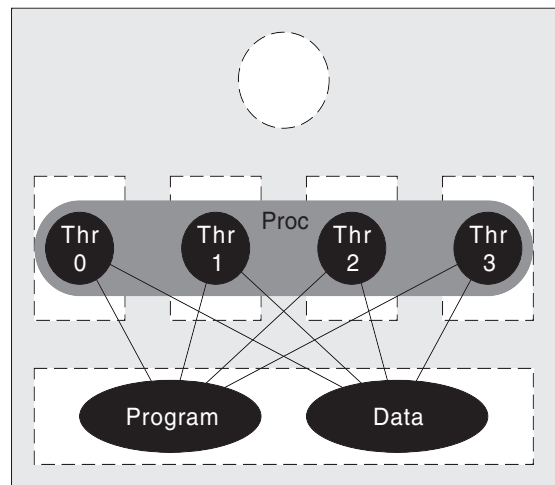
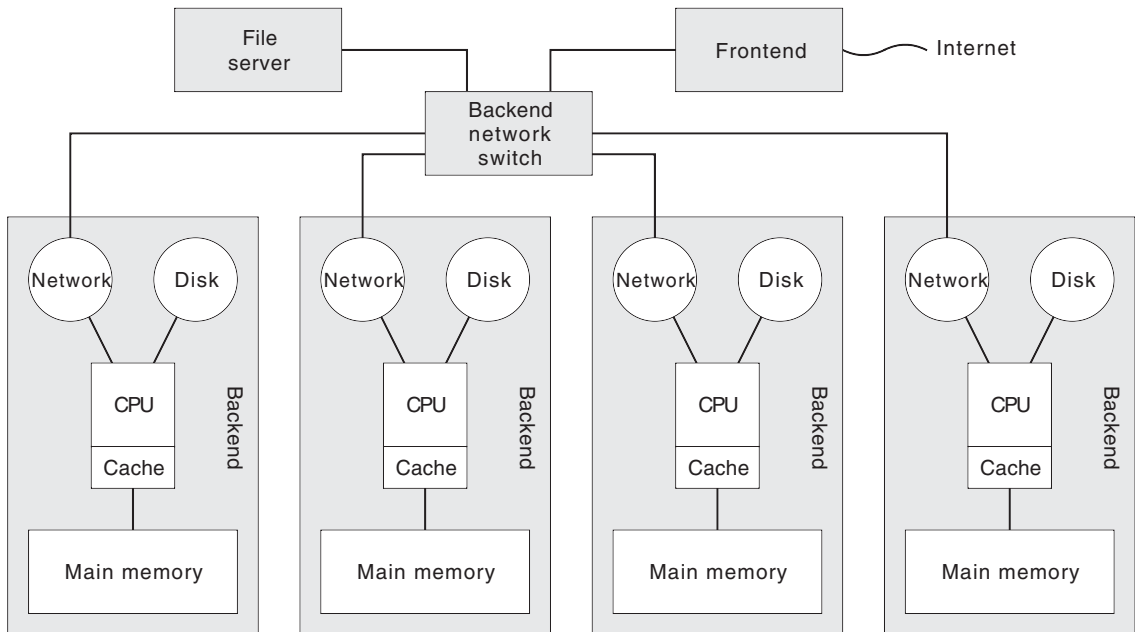


Figure 2.7 SMP parallel program

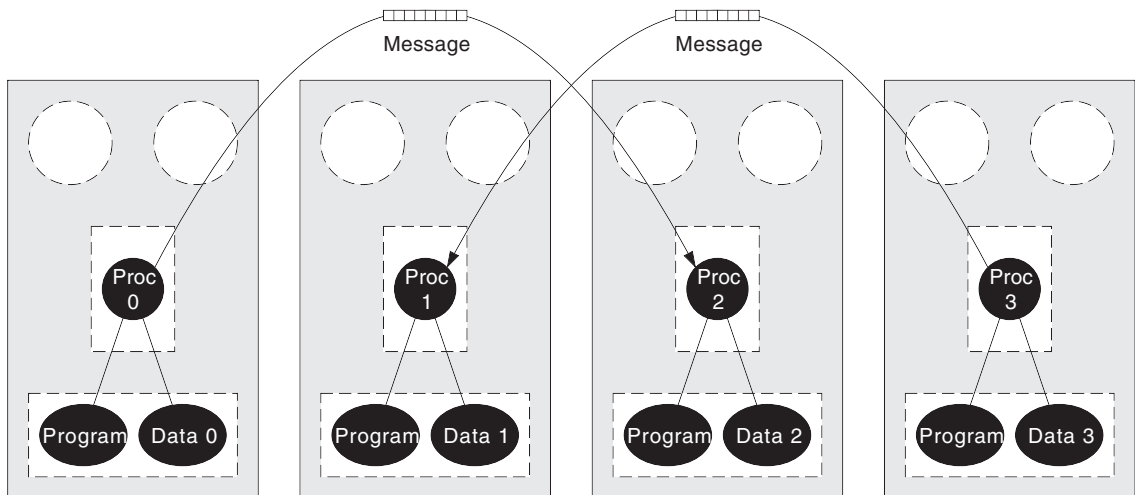
## 2.4 Cluster Parallel Computers

A **cluster** parallel computer consists of multiple interconnected processor nodes (Figure 2.8). There are several **backend processors** that carry out parallel computations. There is also typically a separate **frontend processor**; users log into the frontend to compile and run their programs. There may be a shared file server. Each backend has its own CPU, cache, main memory, and peripherals, such as a local disk drive. Each processor is also connected to the others through a dedicated high-speed **backend network**. The backend network is used only for traffic between the nodes of the cluster; other network traffic, such as remote logins over the Internet, goes to the frontend. Unlike an SMP parallel computer, there is no global shared memory; each backend can access only its local memory. The cluster computer is said to have a **distributed memory**.





**Figure 2.8** Cluster parallel computer



**Figure 2.9** Cluster parallel program

A parallel program running on a cluster parallel computer (Figure 2.9) consists of multiple processes, one process executing on each backend processor. Each process has its own, separate address space. All processes execute the same program, a copy of which resides in each process's address space in each backend's main memory. The program's data is divided into pieces; a different piece resides in each process's address space in each backend's main memory. Each process performs its portion of the computation and stores its results in the data structures in its own local memory. If one process needs a

piece of data that resides in another process's address space, the process that owns the data sends a **message** containing the data through the backend network to the process that needs the data. The processes may also exchange messages to coordinate with one another, without transferring data. Unlike an SMP parallel program where the threads can simply access shared data in the one process's address space, in a cluster parallel program, the processes must be explicitly coded to send and receive messages.

To run a parallel program on a cluster, you typically log into the cluster's frontend processor and launch the program in a process on the frontend, like any other program. Under the hood, the frontend process uses a cluster middleware library to start a backend process on each backend processor, load a copy of the program into each backend's memory, initialize connections for message passing through the backend network, and commence execution of each backend process. At program completion, when all the backend processes have terminated, the frontend process also terminates.

For maximum performance, it's not enough to equip a cluster with fast backend processors. It's also important for the cluster's backend network to have three characteristics: low latency, high bandwidth, and high bisection bandwidth.

**Latency** refers to the amount of time needed to start up a message, regardless of the message's size; it depends on the hardware and software protocols used on the network. **Bandwidth**, measured in bits per second, is the rate at which data is transmitted once a message has started. The time to transfer a message is the latency, plus the message size divided by the bandwidth. A small latency and a large bandwidth will minimize each message's transfer time, thus reducing the cluster parallel program's running time.

**Bisection bandwidth**, also measured in bits per second, refers to the total rate at which data can be transferred if half the nodes in the cluster are sending messages to the other half. In other words, the network is split down the middle—bisected—and each node on one side of the split sends data as fast as possible to a different node on the other side of the split. As we will see in Part III, some cluster parallel programs do in fact send messages from half the processes to the other half at the same time. Ideally, in an  $N$ -node cluster, the network's bisection bandwidth would be  $N/2$  times the bandwidth on a single link. Depending on how the backend network is built, however, the bisection bandwidth may be less than the ideal, which in turn may reduce the cluster parallel program's performance.

Several commodity off-the-shelf technologies are used for backend networks in cluster parallel computers. The available alternatives fall into two categories: Ethernet, and everything else.

**Ethernet**, due to its ubiquity, is the least-expensive alternative. Ethernet interface cards, switches, and cables that support a bandwidth of 1 gigabit per second (1 Gbps), or  $1 \times 10^9$  bits per second, are readily available. Although more expensive, 10 Gbps Ethernet equipment is also available, and 100 Gbps Ethernet is on the horizon. An Ethernet *switch* usually has a large bisection bandwidth. (An Ethernet *hub* does not; you should never use a hub to build a cluster backend network.) Platform-independent programs that communicate over Ethernet are easily written using the standard socket application program interface (API) and the Internet standard TCP/IP protocols. However, Ethernet has a much higher latency—around 150 microseconds ( $150 \mu\text{sec}$ ), or  $150 \times 10^{-6}$  seconds—than the alternatives, especially when using TCP/IP.

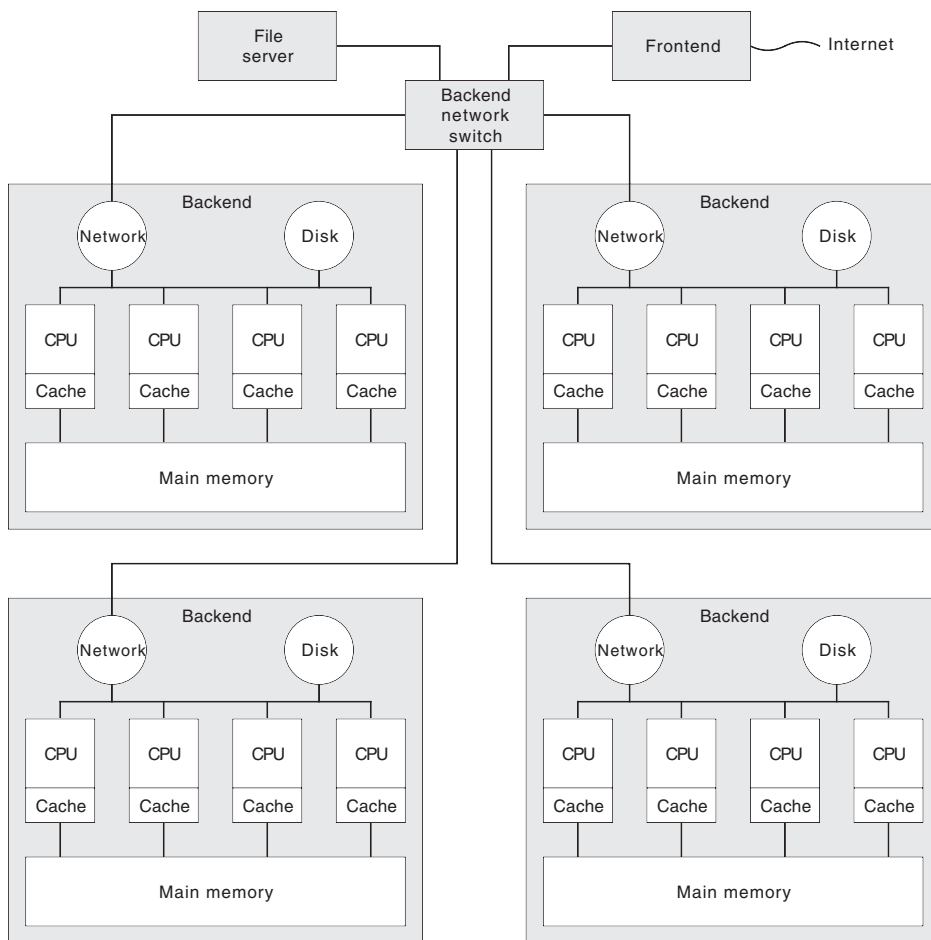
Other interconnection technologies used in cluster parallel computers, such as **InfiniBand**, **Scalable Coherent Interface (SCI)**, and **Myrinet**, are all more or less the same in their gross characteristics. They all support higher bandwidths than Ethernet (up to about 100 Gbps), much lower latencies than Ethernet (in the single microsecond range), and high bisection bandwidths. However, not being nearly as widespread as Ethernet, they are all more expensive. They also tend to require the use of technology-specific software libraries to achieve their full performance. While TCP/IP can be layered on top of these

technologies, thus allowing platform-independent programs to run on these technologies, layering TCP/IP increases the latency due to the TCP/IP protocol overhead.

Ultra-high-performance parallel supercomputers often use interconnect technologies such as InfiniBand, SCI, or Myrinet because of their higher bandwidth and lower latency. Mundane parallel computers usually use Ethernet because of its lower cost and platform-independent programming support.

## 2.5 Hybrid Parallel Computers

A **hybrid** parallel computer is a cluster in which each backend processor is an SMP machine (Figure 2.10). In other words, it is a combination, or hybrid, of cluster and SMP parallel computers. A hybrid parallel computer has both shared memory (within each backend) and distributed memory (between backends). Eventually, all commodity clusters will be hybrid parallel computers because multicore PCs are becoming popular and single-core PCs are becoming harder to find.



**Figure 2.10** Hybrid parallel computer

A hybrid parallel computer is programmed using a combination of cluster and SMP parallel programming techniques (Figure 2.11). Like a cluster parallel computer, each backend runs a separate process with its own address space. Each process executes a copy of the same program, and each process has a portion of the data. Like an SMP parallel computer, each process in turn has multiple threads, one thread running on each CPU. Threads in the same process share the same address space and can access their own shared data structures directly. Threads in different processes must send messages to each other to transfer data.

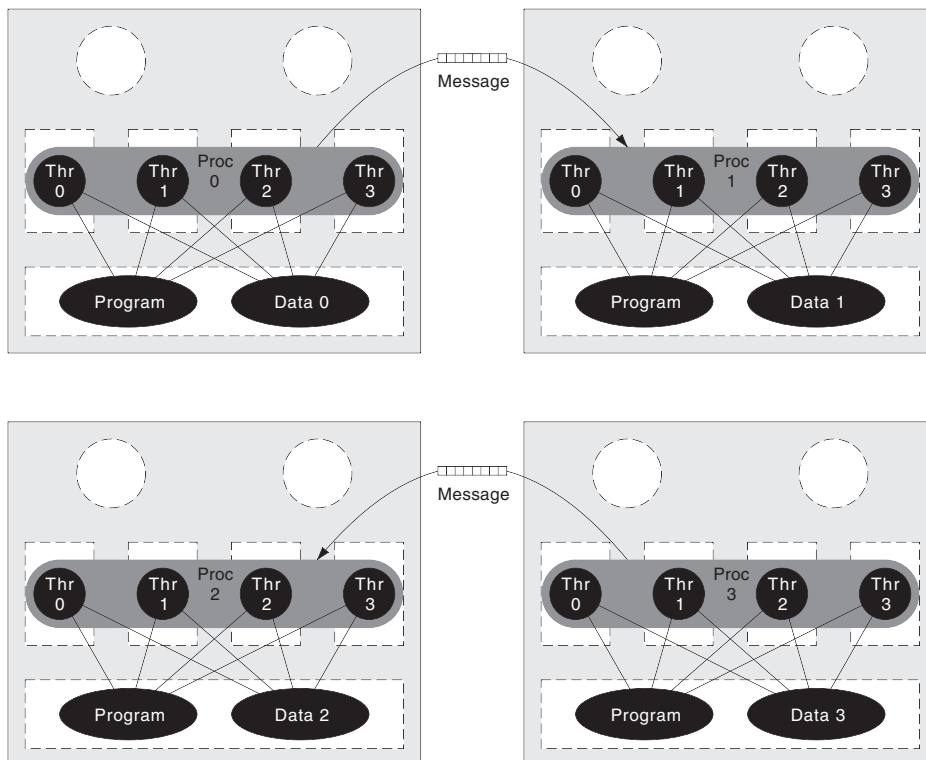
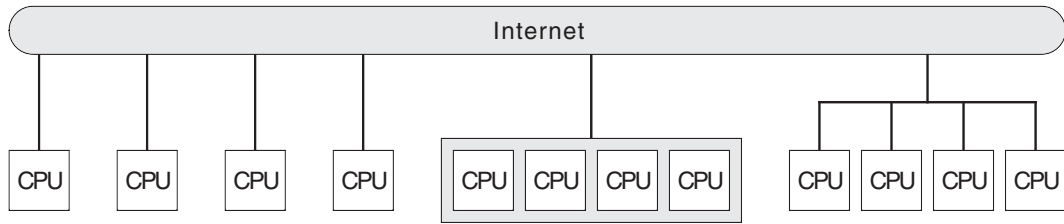


Figure 2.11 Hybrid parallel program

## 2.6 Computing Grids

Architecturally, a **computing grid** is the same as a cluster parallel computer, except that the processors are not all located in the same place and are not all connected to a common, dedicated backend network. Instead, the processors are located at diverse sites and are connected through a combination of local area networks and the Internet (Figure 2.12).

Often, a grid is set up by a consortium of companies, universities, research institutions, and government agencies—the member organizations contributing computers to the grid. The Open Science Grid (OSG), for example, is a grid devoted to large-scale scientific computation, with thousands of processors located at 85 institutions in Brazil, Canada, England, Germany, Korea, Taiwan, and the United States.



**Figure 2.12** Computing grid with single processors, an SMP, and a cluster

Other grids are based on “volunteer computing.” Users download a special client program to their desktop PCs. The client program runs as a low-priority background process, or sometimes as a screen saver. When the PC becomes idle, the client program starts up and executes a portion of a parallel computation, communicating with other nodes over the Internet. Projects using volunteer computing grids include SETI@home, the Great Internet Mersenne Prime Search (GIMPS), and dozens of others. The Berkeley Open Infrastructure for Network Computing (BOINC) is a general framework for developing parallel programs that run on volunteer computing grids.

Computing grids are programmed in the same way as cluster parallel computers, with multiple processes running on the various grid machines. However, a parallel program that performs well on a cluster is not necessarily well suited for a grid. The Internet’s latency is orders of magnitude larger, and the Internet’s bandwidth is orders of magnitude smaller, than a typical cluster backend network. Because some of the computers running the grid program may be connected through the Internet, the average message takes a lot longer to send on a grid than on a cluster. Thus, parallel programs that require intensive message passing do not perform well on a grid. Problems best suited for a grid are those that can be divided into many pieces that are computed independently with little or no communication.

In this book, we will study how to build parallel programs for *tightly coupled* processors: SMP parallel computers where all processors use a single shared memory; and cluster and hybrid parallel computers where all processors are connected to the same high-speed backend network. Parallel programming for *loosely coupled* computing grids is beyond the scope of this book.

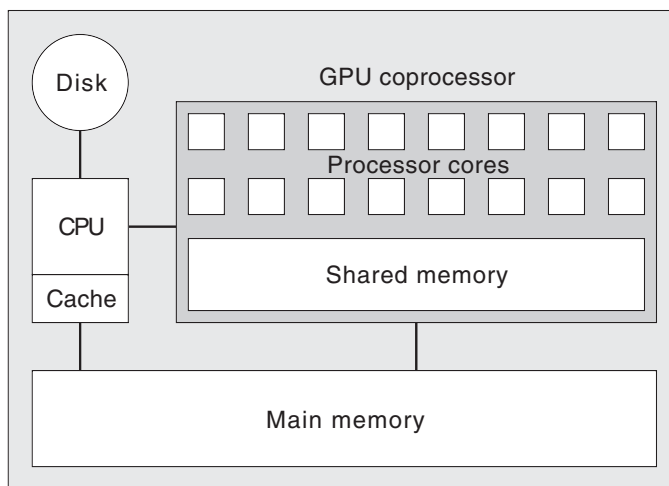
## 2.7 GPU Coprocessors

CPU, memory, and Ethernet chips are not the only chips that have become commodities. Driven by the market’s insatiable appetite for ever-higher-performing graphics displays on PCs and game consoles, **graphics processing unit (GPU)** chips have also become commonplace.

Acting as a **coprocessor** to the main CPU, the GPU is a specialized chip that handles graphics rendering calculations at very high speeds. The CPU sends high-level commands to the GPU to draw lines and fill shapes with realistic shading and lighting; the GPU then calculates the color of each pixel and drives the display. Because the pixels can be computed independently, the GPU typically has multiple processing cores and calculates multiple pixels in parallel.

Programmers have realized that GPUs can be used to do parallel computations other than pixel rendering, and have even coined an acronym for it: **GPGPU**, or General Purpose computation on Graphics Processing Units. In response, GPU vendors have repackaged their graphics cards as general-purpose massively parallel coprocessors, complete with on-board shared memory and with APIs for parallel programming on the GPU. A GPU coprocessor card transforms a regular PC into what marketers call a

“desktop supercomputer” (Figure 2.13). The low cost of the commodity GPU chips makes a GPU coprocessor an attractive alternative to general-purpose multicore CPUs and clusters.



**Figure 2.13** Parallel computer with GPU coprocessor

A GPU’s instruction set is usually more limited than a regular CPU’s. For example, a GPU may support single-precision floating-point arithmetic, but not double precision. Also, the GPU’s cores typically must all perform identical operations simultaneously, each core operating on its own data items. For these reasons, current GPU coprocessors cannot run arbitrary parallel programs. However, GPU coprocessors excel at running “inner loops,” where the same statements are performed on every element of an array or matrix at very high speed in parallel. Thus, a GPU parallel program usually consists of regular code executed on the main CPU with a computation-intensive section executed on the GPU. Unfortunately, space limitations do not permit covering GPU parallel programming in this book.

## 2.8 SMPs, Clusters, Hybrids: Pros and Cons

Why are there three prevalent parallel computer architectures? Why not use the same architecture for all parallel computers? The reason is that each architecture is best suited for certain kinds of problems and is not well suited for other kinds of problems.

An SMP parallel computer is well suited for a problem where there are data dependencies between the processors—where each processor produces results that are used by some or all of the other processors. By putting the data in a common address space (shared memory), each thread can access every other thread’s results directly at the full speed of the CPU and memory. A cluster parallel computer running such a program would have to send many messages between the processors. Because sending a message is orders of magnitude slower than accessing a shared memory location, even with high-speed interconnects such as InfiniBand, SCI, and Myrinet, an SMP parallel computer would outperform a cluster parallel computer on this problem.

Conversely, a cluster parallel computer is well suited for a problem where there are few or no data dependencies between the processors—where each processor can compute its own results with little or no communication with the other processors. The more communication needed, the poorer a cluster parallel program will perform compared to an SMP parallel program.

On the other hand, two limitations are encountered when trying to scale up to larger problem sizes on an SMP parallel computer. First, there is a limit on the physical memory size. A 32-bit CPU can access at most 4 gigabytes ( $2^{32}$  bytes) of physical memory. While a 64-bit CPU can theoretically access up to 16 million terabytes ( $2^{64}$  bytes) of physical memory, it will be a long time before any single CPU has a physical memory that large. Although virtual memory lets a process access a larger address space than physical memory, the performance of a program whose instructions and data do not fit in physical memory would be severely reduced due to swapping pages back and forth between physical memory and disk. Second, there is a limit on the number of CPUs that can share the same main memory. The more CPUs there are, the more circuitry is needed to coordinate the memory transactions from all the CPUs and to keep all the CPUs' caches coherent. Thus, an SMP parallel program's problem size cannot scale up past the point where its data no longer fits in main memory, and its speedup or sizeup cannot scale up past a certain number of CPUs.

With a cluster parallel computer, there are no limits on scalability due to main memory size or number of processors. On a  $K$ -node cluster, the maximum total amount of memory is  $K$  times the maximum on one node. Thus, if you need more memory, more speedup, or more sizeup, just add more nodes to the cluster. However, because message latency tends to increase as the number of nodes on the network increases, causing program performance to decrease, a cluster cannot keep growing forever. Still, commodity clusters with hundreds and even thousands of nodes have been built; the largest commodity SMPs have dozens of nodes at most.

Like a cluster, a hybrid parallel computer can be scaled up to larger problem sizes by adding more nodes. In addition, a hybrid parallel computer is especially well suited for a problem that can be broken into chunks having few or no data dependencies between chunks, but that can have significant data dependencies within each chunk. The chunks can be computed in parallel by separate processes running on the cluster nodes and passing messages among each other. Within each chunk, the results can be computed in parallel by separate threads running on the node's CPUs and accessing data in shared memory. Many of the supercomputers in the TOP500 List are commodity hybrid parallel computers.

Any parallel program can be implemented to run on an SMP, cluster, or hybrid parallel computer. (For some of the example programs in this book, we will look at all three variations.) Which kind of parallel computer, then, should you use to solve your high-performance computing problem—assuming your local computer center even offers you a choice? The answer is to use the kind of parallel computer that gives the best performance on your problem. However, because so many factors influence performance, the only way to know for sure is to implement the appropriate versions of the program and try them on the available parallel computers. As we study how to design and code parallel programs, we will also discuss how the program's design influences the program's performance, and we will see how certain kinds of parallel programs perform better on certain kinds of parallel computers.

## 2.9 Parallel Programming Libraries

It is perfectly possible to write parallel programs using a standard programming language and generic operating system kernel functions. You can write a multithreaded program in C using the standard POSIX thread library (Pthreads), or in Java using Java's built-in Thread class. If you run this program on an SMP parallel computer, each thread will run on a different processor simultaneously, yielding a parallel speedup. You can write a multiprocess program in C using the standard socket API to communicate between processes, or in Java using Java's built-in java.net.Socket class. If you run copies of this process on the backend processors of a cluster parallel computer, the simultaneously running processes will yield a parallel speedup.

However, parallel programs are generally not written this way, for two reasons. First, some programming languages popular in domains that benefit from parallel programming—such as Fortran for scientific computing—do not support multithreaded programming and network programming as well as other languages. Second, using low-level thread and socket libraries increases the effort needed to write a parallel program. It takes a great deal of effort to write the code that sets up and coordinates the multiple threads of an SMP parallel program, or to write the code that sets up network connections and sends and receives messages for a cluster parallel program. Parallel program users are interested in solving problems in their domains, such as searching a massive DNA sequence database or calculating the motion of stars in a galaxy, not in writing thread or network code. Indeed, many parallel program users may lack the expertise to write thread or network code.

Instead, to write a parallel program, you use a **parallel programming library**. The library encapsulates the low-level thread or network code that is the same in any parallel program, presenting you with easy-to-use, high-level parallel programming abstractions. You can then devote most of your parallel programming effort to solving your domain problem using those abstractions.

**OpenMP** is the standard library for SMP parallel programming. OpenMP supports the Fortran, C, and C++ languages. By inserting special OpenMP **pragmas** into the source code, you designate which sections of code are to be executed in parallel by multiple threads. You then run the annotated source code through a special OpenMP compiler. The OpenMP compiler looks at the OpenMP pragmas, *rewrites* your source code to add the necessary low-level threading code, and compiles your now-multithreaded program as a regular Fortran, C, or C++ program. You then run your program as usual on an SMP parallel computer. See Appendix A for further information about OpenMP.

The **Message Passing Interface (MPI)** is the standard library for cluster parallel programming. Like OpenMP, MPI supports the Fortran, C, and C++ languages. Unlike OpenMP, MPI requires no special compiler; it is just a subroutine library. You write your parallel program like any regular program, calling the MPI library routines as necessary to send and receive messages, and compile your program as usual. To run your program on a cluster parallel computer, you execute a special MPI **launcher** program. The launcher takes care of starting a process to run your compiled executable program on each backend processor. The MPI library routines your program calls then take care of all the details of setting up network connections between processes and passing messages back and forth. See Appendix B for further information about MPI.

As already mentioned, hybrid parallel computers are becoming popular, due to the wide availability of multicore PCs. However, as yet, there are no standard libraries for hybrid parallel programming. OpenMP has no routines for message passing. MPI has no capabilities for executing sections of code



in parallel in multiple threads. Writing a hybrid parallel program using both OpenMP and MPI is not guaranteed to work, because the MPI standard does not require all MPI implementations to support multithreaded programs (although an MPI implementation is allowed to do so). While hybrid parallel programs can be written solely using MPI by running a separate *process* (rather than a thread) on each CPU of each node, sending messages between different processes' address spaces on the same node often yields poorer performance than simply sharing the same address space among several threads.

In addition, neither the official OpenMP standard nor the official MPI standard supports the Java language. Yet Java is becoming the language that most computing students learn. While several unofficial versions of MPI and OpenMP in Java have appeared, none can be considered a standard, and none are designed for hybrid parallel programming.

In this book, we will use the **Parallel Java Library** to learn how to build parallel programs. Parallel Java includes both the multithreaded parallel programming capabilities of OpenMP and the message-passing capabilities of MPI, integrated in a single library. Thus, Parallel Java is well suited for hybrid parallel programming as well as SMP and cluster parallel programming. Parallel Java also includes its own middleware for running parallel programs on a cluster. Because the library is written in 100% Java, Parallel Java programs are portable to any machine that supports Java (JDK 1.5). Appendices A and B compare and contrast Parallel Java with OpenMP and MPI.

## 2.10 For Further Information

On the history of parallel computers:

- G. Wilson. A chronology of major events in parallel computing. University of Toronto Computer Systems Research Institute Technical Report CSRI-312, December 1994. <ftp://ftp.cs.toronto.edu/csrg-technical-reports/312/csri312.ps>

On Beowulf:

- T. Sterling, D. Becker, D. Savarese, J. Dorband, U. Ranawake, and C. Packer. Beowulf: A parallel workstation for scientific computation. In *Proceedings of the 24th International Conference on Parallel Processing (ICPP 1995)*, 1995, volume 1, pages 11–14.
- D. Ridge, D. Becker, P. Merkey, and T. Sterling. Beowulf: harnessing the power of parallelism in a Pile-of-PCs. In *Proceedings of the 1997 IEEE Aerospace Conference*, 1997, volume 2, pages 79–91.
- Beowulf.org Web site. <http://www.beowulf.org/>

On the Stone SouperComputer (and the tale of “Stone Soup”):

- W. Hargrove, F. Hoffman, and T. Sterling. The do-it-yourself supercomputer. *Scientific American*, 265(2):72–79, August 2001.
- The Stone SouperComputer. <http://www.extremelinux.info/stonesoup/>

On the Parallel Virtual Machine (PVM) library:

- V. Sunderam. PVM: a framework for parallel distributed computing. *Concurrency Practice and Experience*, 2(4):315–339, December 1990.
- A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine: A Users' Guide and Tutorial for Networked Parallel Computing*. MIT Press, 1994.
- PVM: Parallel Virtual Machine.  
[http://www.csm.ornl.gov/pvm/pvm\\_home.html](http://www.csm.ornl.gov/pvm/pvm_home.html)

On the official MPI standard:

- Message Passing Interface Forum Web Site.  
<http://www.mpi-forum.org/>
- Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*. June 12, 1995. (MPI Version 1.1)  
<http://www.mpi-forum.org/docs/mpi-11.ps>
- Message Passing Interface Forum. *MPI-2: Extensions to the Message-Passing Interface*. July 18, 1997. (MPI Version 2.0)  
<http://www.mpi-forum.org/docs/mpi-20.ps>

A comparison of PVM and MPI:

- G. Geist, J. Kohl, and P. Papadopoulos. PVM and MPI: a comparison of features. *Calculateurs Paralleles*, 8(2):137–150, 1996.

On the official OpenMP standard:

- OpenMP.org Web Site. <http://openmp.org/wp/>
- OpenMP Architecture Review Board. *OpenMP Application Program Interface, Version 3.0*. May 2008.  
<http://www.openmp.org/mp-documents/spec30.pdf>

On the TOP500 supercomputer list and the LINPACK benchmark:

- TOP500 Supercomputer Sites. <http://www.top500.org/>
- LINPACK. <http://www.netlib.org/linpack/>
- LINPACK Benchmark—Java Version.  
<http://www.netlib.org/benchmark/linpackjava/>

On the Open Science Grid:

- Open Science Grid. <http://www.opensciencegrid.org/>

On volunteer computing grids:

- Distributed Computing Projects directory.  
<http://www.distributedcomputing.info/>
- SETI@home. <http://setiathome.berkeley.edu/>
- GIMPS. <http://www.mersenne.org/>
- Berkeley Open Infrastructure for Network Computing.  
<http://boinc.berkeley.edu/>

On parallel computing with GPUs:

- GPGPU Web Site. <http://www.gpgpu.org/>
- N. Goodnight, R. Wang, and G. Humphreys. Computation on programmable graphics hardware. *IEEE Computer Graphics and Applications*, 25(5):12–15, September/October 2005.
- J. Owens, M. Houston, D. Luebke, S. Green, J. Stone, and J. Phillips. GPU computing. *Proceedings of the IEEE*, 96(5):879–899, May 2008.
- R. Fernando, editor. *GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics*. Pearson Education, 2004.
- M. Pharr, editor. *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*. Pearson Education, 2005.
- H. Nguyen, editor. *GPU Gems 3*. Addison-Wesley, 2007.

On the Parallel Java Library:

- A. Kaminsky. Parallel Java: a unified API for shared memory and cluster parallel programming in 100% Java. In *Proceedings of the 21st IEEE International Parallel and Distributed Processing Symposium (IPDPS 2007)*, March 2007.
- Parallel Java Library (including downloads).  
<http://www.cs.rit.edu/~ark/pj.shtml>
- Parallel Java documentation.  
<http://www.cs.rit.edu/~ark/pj/doc/index.html>

*This page intentionally left blank*

# How to Write Parallel Programs

in which we discover the three principal patterns for designing parallel programs; we encounter examples of problems for which each pattern is suited; and we see how to realize the patterns on practical parallel computers

## 3.1 Patterns of Parallelism

Let's say you are given a problem that requires a massive amount of computation—such as finding, in an enormous genomic database of DNA sequences, the few sequences that best match a short query sequence; or calculating the three-dimensional positions as a function of time of a thousand stars in a star cluster under the influence of their mutual gravitational forces. To get the answer in an acceptable amount of time, you need to write a parallel program to solve the problem. But where do you start? How do you even think about designing a parallel program?

In an 1989 paper titled “How to write parallel programs: a guide to the perplexed,” Nicholas Carriero and David Gelernter of Yale University addressed this question by codifying three patterns for designing parallel programs: **result parallelism**; **agenda parallelism**; and **specialist parallelism**. Each pattern encompasses a whole class of similarly structured problems; furthermore, each pattern suggests how to design a parallel program for any problem in that class. Using the patterns, the steps for designing a parallel program are the following:

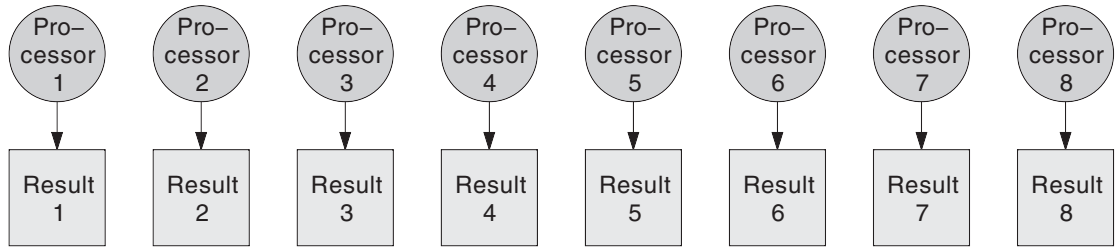
- Identify the pattern that best matches the problem.
- Take the pattern's suggested design as the starting point.
- Implement the design using the appropriate constructs in a parallel programming language.

Next, we describe each of the three patterns and give examples of how they are used.

## 3.2 Result Parallelism

In a problem that exhibits **result parallelism** (Figure 3.1), there is a collection of multiple results. The individual results are all computed in parallel, each by its own processor. Each processor is able to carry out the complete computation to produce one result. The conceptual parallel program design is:

Processor 1:	Compute result 1
Processor 2:	Compute result 2
...	
Processor $N$ :	Compute result $N$



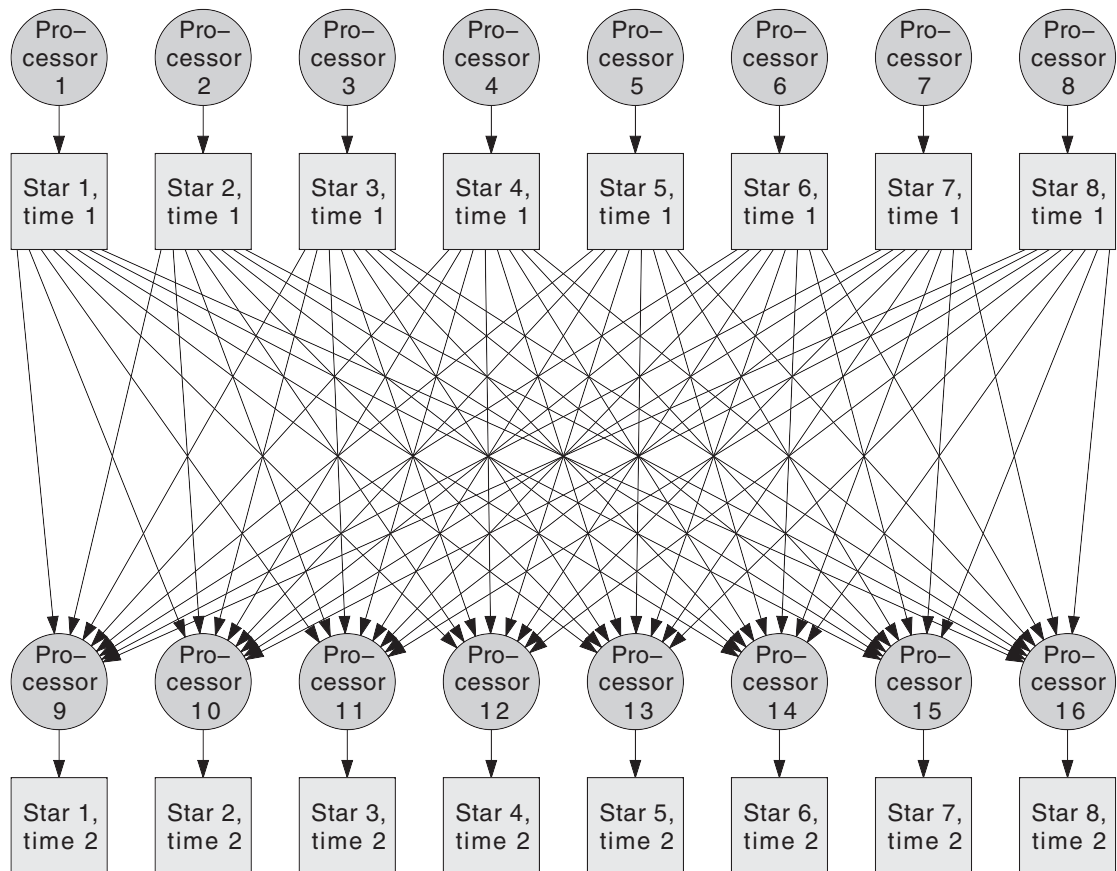
**Figure 3.1** Result parallelism

A problem that requires computing each element of a data structure often exhibits result parallelism. As an example, consider the problem of calculating all the pixels in all the frames of a computer-animated film. One way to solve the problem is to assign a separate processor to calculate each pixel. Another way is to assign a separate processor to render each entire frame, calculating all the pixels in that frame. The former is an example of **fine-grained parallelism**, where each result requires a small amount of computation. The latter is an example of **coarse-grained parallelism**, where each result requires a large amount of computation. In both cases, though, none of the processors needs to use the result calculated by any other processor; there are no dependencies among the computations. Thus, in concept, all the processors can start at the same time, calculate, and finish at the same time.

Other problems, however, do have dependencies among the computations. Consider calculating the 3-D positions of  $N$  stars in a star cluster as a function of time for a series of  $M$  time steps. The result can be viewed as an  $M \times N$ -element matrix of positions: row 1 contains the stars' positions after the first time step; row 2 contains the stars' positions after the second time step; and so on. In concept, the parallel program has  $M \times N$  processors. The processors in row 1 can begin computing their results immediately. Each processor in row 1 calculates the gravitational force on its star due to each of the other stars, using the stars' input initial positions. Each processor in row 1 then moves its star by one time step in a direction determined by the net gravitational force, and the star's new position becomes the processor's result. However, the processors in row 2 cannot begin computing their results immediately. To do their computations, these processors need the stars' positions after the first time step, so these processors must wait until all the row-1 processors have computed their results. We say there are **sequential dependencies** from the computations in each row to the computations in the next row (Figure 3.2). There are no sequential dependencies between the computations in the same row, though.

Faced with the problem of calculating stellar motion for, say, one thousand stars and one million time steps, you might wonder where to find a parallel computer with enough hardware to compute each element of the result in its own separate processor. Keep in mind that, for now, we are still in the realm of *conceptual* parallel program design, where there are no pesky hardware limitations. In Section 3.5 we will see how to translate this conceptual design into a real parallel program.

Recalculating a spreadsheet is another example of a result parallel problem with sequential dependencies. The spreadsheet cell values are the results computed by the program. Conceptually, each cell has its own processor that computes the value of the cell's formula. When you type a new value into an input cell, the processors all calculate their respective cells' formulas. Normally, all the cells can be calculated in parallel. However, if the formula for cell B1 uses the value of cell A1, then the B1 processor must wait until the A1 processor has finished. Soon all spreadsheets will have to be result parallel programs to get full performance out of a desktop or laptop computer's multicore CPU chip.



**Figure 3.2** Result parallelism with sequential dependencies—star cluster simulation

### 3.3 Agenda Parallelism

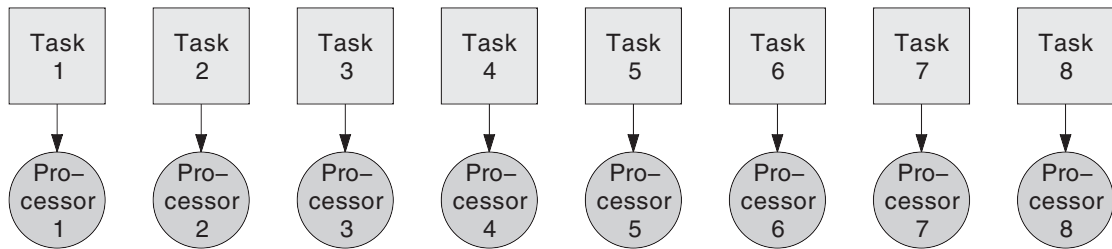
In a problem that exhibits **agenda parallelism** (Figure 3.3), there is an agenda of tasks that must be performed to solve the problem, and there is a team of processors, each processor able to perform any task. The conceptual parallel program design is:

```

Processor 1:    Perform task 1
Processor 2:    Perform task 2
...
Processor N:    Perform task N
  
```

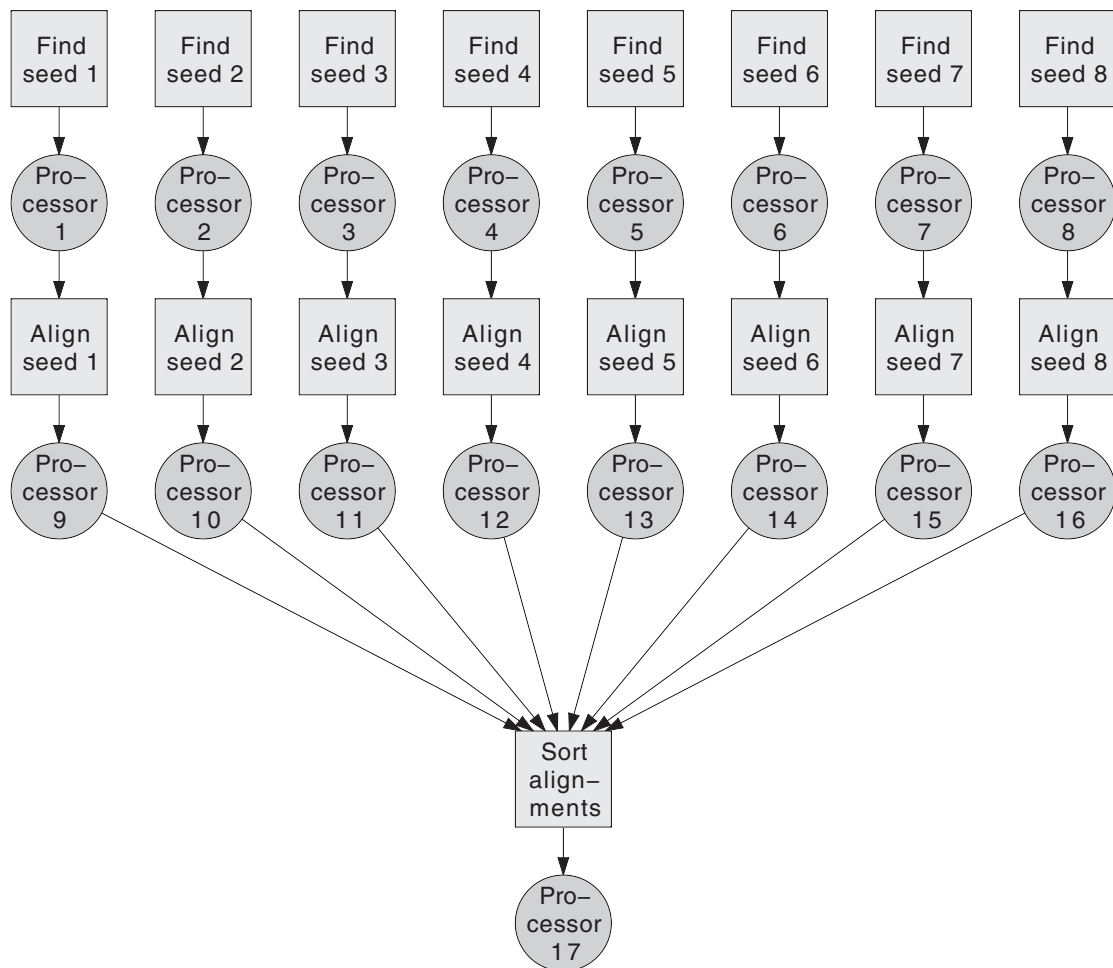


A problem that requires computing one result, or a small number of results, from a large number of inputs often exhibits agenda parallelism. Querying a DNA sequence database is one example. The agenda items are: “Determine if the query sequence matches database sequence 1”; “Determine if the query sequence matches database sequence 2”; and so on. Each of these tasks can be performed independently of all the others, in parallel.



**Figure 3.3** Agenda parallelism

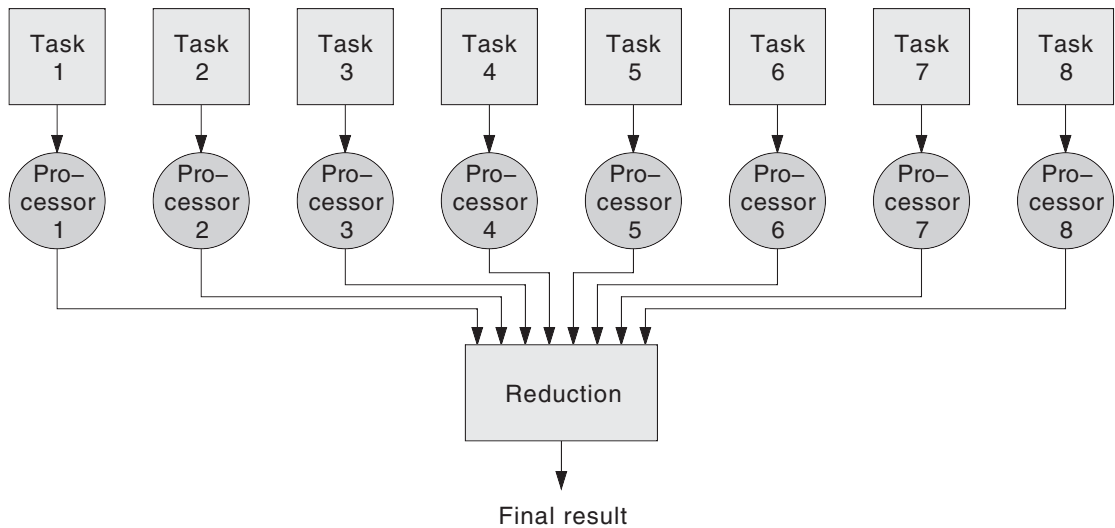
Other agenda parallel problems have sequential dependencies among the tasks (Figure 3.4). Certain tasks cannot start until other tasks have finished. The Basic Local Alignment Search Tool (BLAST) program, a widely used DNA and protein sequence database search program, can be viewed as an agenda parallel problem. BLAST proceeds in a series of phases. In the first phase, BLAST looks for matches between short pieces of the query sequence and short pieces of the sequence database; this results in a large number of tentative starting points known as “seeds.” In the second phase, BLAST takes each seed and tries to align the complete query sequence with the sequence database, starting from the seed’s location. BLAST computes a score for each alignment that tells how biologically plausible the alignment is; alignments that don’t result in a good match (too low a score) are discarded. In the third phase, BLAST sorts the surviving alignments into descending order of plausibility and outputs the alignments, most plausible first. Conceptually, the phase-1 agenda items are of the form “For seed *X*, match piece *Y* of the query against piece *Z* of the database.” These can all be done in parallel. The phase-2 agenda items are of the form “Align the query with the database at seed *X*’s location and compute the plausibility score.” These can all be done in parallel with each other, but each must wait until the corresponding phase-1 agenda item has finished. The final agenda item, “Sort and output the alignments,” must wait until the phase-2 agenda items have finished.



**Figure 3.4** Agenda parallelism with sequential dependencies—BLAST

A result parallel problem could be viewed as an agenda parallel problem, where the agenda items are “Compute result 1,” “Compute result 2,” and so on. The difference is that in a result parallel problem, we are typically interested in *every* processor’s result. In an agenda parallel problem, we are typically not interested in every processor’s (agenda item’s) result, but only in certain results, or only in a combination or summary of the individual results.

When an agenda parallel program’s output is a combination or summary of the individual tasks’ results, the program is following the so-called **reduction** pattern (Figure 3.5). The number of results is *reduced* from many down to one. Often, the final result is computed by applying a **reduction operator** to the individual results. For example, when the operator is addition, the final result is the sum of the tasks’ results. When the operator is minimum, the final result is the smallest of the tasks’ results.



**Figure 3.5** Agenda parallelism with reduction

## 3.4 Specialist Parallelism

In a problem that exhibits **specialist parallelism** (Figure 3.6), like agenda parallelism, there is a group of tasks that must be performed to solve the problem, and there is a team of processors. But, unlike agenda parallelism, each processor performs only a specific one of the tasks, not just any task. Often, one specialist processor's job is to perform the same task on a series of items. The conceptual parallel program design is:

Processor 1:	For each item:
	Perform task 1 on the item
Processor 2:	For each item:
	Perform task 2 on the item
...	
Processor $N$ :	For each item:
	Perform task $N$ on the item

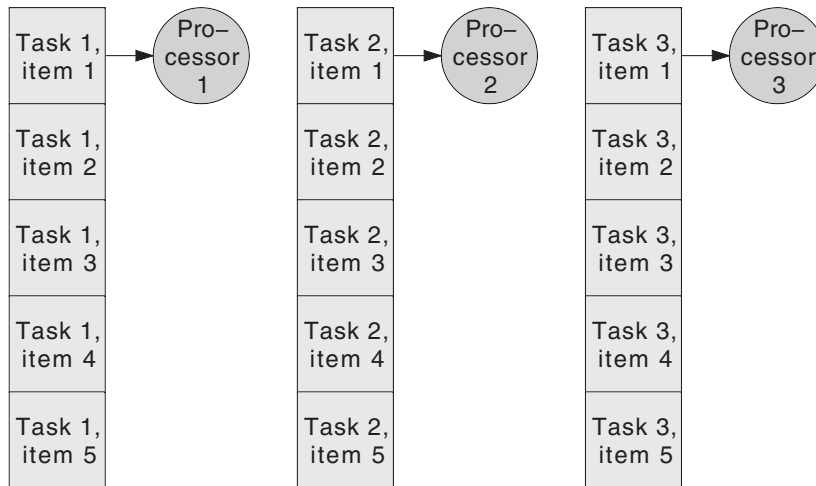
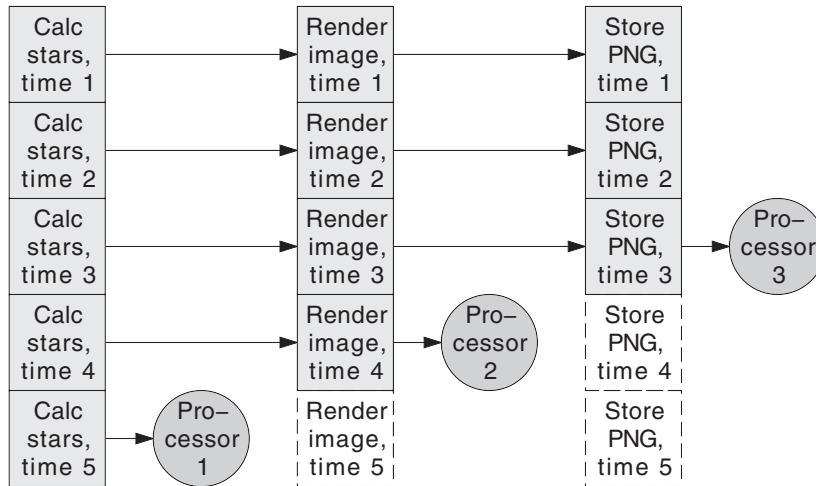


Figure 3.6 Specialist parallelism

When there are sequential dependencies between the tasks in a specialist parallel problem, the program follows the so-called **pipeline** pattern. The output of one processor becomes the input for the next processor. All the processors execute in parallel, each taking its input from the preceding processor's previous output.

Consider again the problem of calculating the 3-D positions of  $N$  stars in a star cluster as a function of time for a series of  $M$  time steps. Now add a feature: At each time step, the program must create an image of the stars' positions and store the image in a Portable Network Graphics (PNG) file. This problem can be broken into three steps: calculate the stars' positions; create an image of the star's positions; and store the image in a PNG file. Each step requires a certain amount of computation: to calculate the numerical  $(x,y,z)$  coordinates of each star; to determine the color of each pixel so as to display the stars' 3-D positions properly in the 2-D image; and to compress the pixel data and store it in a PNG file. The three steps can be performed in parallel by three processors in a specialist parallel program (Figure 3.7). While one processor is calculating the stars' positions for time step  $t$ , another processor is taking the stars' positions for time step  $t-1$  and rendering an image, and a third processor is taking the image for time step  $t-2$ , compressing it, and storing it in a file. A program like this, where some processors are doing computations and other processors are doing file input or output, is said to be using the **overlapping** pattern, also called the **overlapped computation and I/O** pattern.



**Figure 3.7** Specialist parallelism with sequential dependencies—star cluster simulation

It's possible for a problem to exhibit multiple patterns of parallelism. The star cluster program, for example, can combine result parallelism with specialist parallelism. At each time step, we can have  $N$  processors each calculating one star's position (result parallelism); one processor rendering the image for the previous time step (specialist parallelism); and one processor compressing and writing the previous image to a PNG file (specialist parallelism). Putting it another way, the specialist parallel task of computing the stars' positions for one time step is itself a subproblem that exhibits result parallelism.

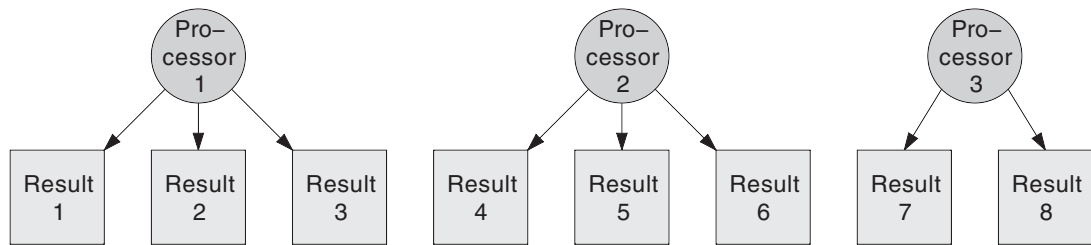
To sum up the three patterns: Result parallelism focuses on the results that can be computed in parallel. Agenda parallelism focuses on the tasks that can be performed in parallel. Specialist parallelism focuses on the processors that can execute in parallel.

## 3.5 Clumping, or Slicing

Applying the parallel program design patterns, as described so far, to a problem large enough to need a parallel computer would require a veritable horde of processors. A result parallel problem with a billion results would require a billion processors, one to compute each result. An agenda parallel problem with a billion agenda items would require a billion processors as well. (Specialist parallel problems tend not to require such large numbers of *different* specialists.) A problem size of one billion— $10^9$ , or about  $2^{30}$ —is by no means far-fetched. We will run even the simple, pedagogical parallel programs in this book on problems of this size. Real-world parallel programs regularly run on much larger problems.

The difficulty, of course, is finding a parallel computer with billions and billions of processors. Well-funded government or academic high-performance computing centers may have parallel computers with processors numbering in the thousands. Most of us would count ourselves lucky to have a dozen or two.

To fit a large parallel problem on an actual parallel computer with comparatively few processors, we use **clumping**. Many conceptual processors are clumped together and executed by one actual processor. In a result parallel program, each processor computes a clump of many results instead of just one result (Figure 3.8).



**Figure 3.8** Result parallelism with clumping (or slicing)

**Slicing** is another way of looking at the same thing. Rather than thinking of clumping many processors into one, think of dividing the result data structure into slices, as many slices as there are processors, and assigning one processor to compute all the results in the corresponding slice. For example, suppose there are 100 results and 4 processors. The design of the result parallel program with slicing is:

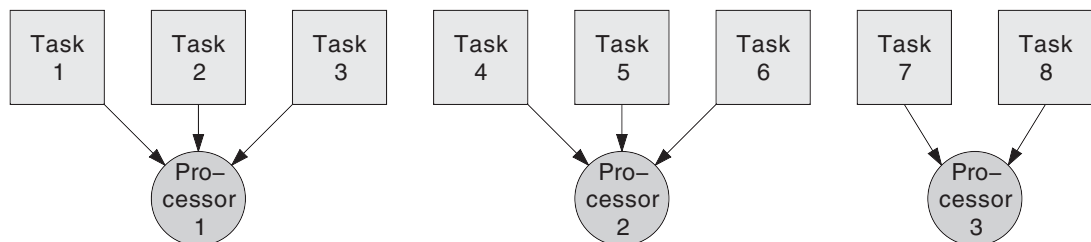
Processor 1:	Compute result 1, 2, . . . 24, 25
Processor 2:	Compute result 26, 27, . . . 49, 50
Processor 3:	Compute result 51, 52, . . . 74, 75
Processor 4:	Compute result 76, 77, . . . 99, 100

In the rest of the book, we will study several parallel programming constructs that automatically slice up a problem of any size to use however many processors the parallel computer has.

## 3.6 Master-Worker

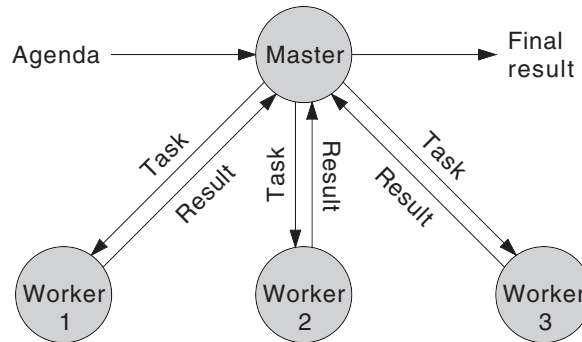
An agenda parallel problem with many more tasks than processors must also use clumping on a real parallel computer (Figure 3.9). Each processor performs many tasks, not just one. Conceptually, the agenda takes the form of a bag of tasks. Each processor repeatedly takes a task out of the bag and performs the task, until the bag is empty, as follows:

Processor 1:	While there are more tasks: Get and perform the next task
Processor 2:	While there are more tasks: Get and perform the next task ...
Processor $K$ :	While there are more tasks: Get and perform the next task



**Figure 3.9** Agenda parallelism with clumping

On a cluster parallel computer, an agenda parallel problem with clumping is often realized concretely using the **master-worker** pattern (Figure 3.10). There is one master processor in charge of the agenda, and there are  $K$  worker processors that carry out the agenda items. The master sends tasks to the workers, receives the task results from the workers, and keeps track of the program's overall results. Each worker receives tasks from the master, computes the task results, and sends the results back to the master. The conceptual parallel program design is:



**Figure 3.10** Agenda parallelism, master-worker pattern

Master:	Send initial task to each worker Repeat: Receive task result from any worker $X$ Record task result Get next task If there are no more tasks, tell worker $X$ to stop Otherwise, send task to worker $X$
Worker 1:	Repeat: Receive a task from the master If there are no more tasks, stop Compute task results Send results to the master
Worker 2:	Repeat: Receive a task from the master If there are no more tasks, stop Compute task results Send results to the master
...	
Worker $K$ :	Repeat: Receive a task from the master If there are no more tasks, stop Compute task results Send results to the master

In the rest of the book, we will study many parallel programs designed to run on SMP parallel computers, cluster parallel computers, and hybrid cluster parallel computers. All these programs, however, will follow one of the three parallel design patterns—result parallelism, agenda parallelism, specialist parallelism—or a combination thereof. Before diving into an in-depth study of the parallel programming constructs that let us implement these patterns, in Chapter 4 we will wet our toes with a small introductory parallel program.

## 3.7 For Further Information

On the three parallel design patterns—Carriero’s and Gelernter’s paper:

- N. Carriero and D. Gelernter. How to write parallel programs: a guide to the perplexed. *ACM Computing Surveys*, 21(3):323–357, September 1989.

Carriero and Gelernter later expanded their paper into a book:

- N. Carriero and D. Gelernter. *How to Write Parallel Programs: A First Course*. MIT Press, 1990.

A more recent book about parallel design patterns, coming from the “patterns movement” in software design:

- T. Mattson, B. Sanders, and B. Massingill. *Patterns for Parallel Programming*. Addison-Wesley, 2005.

On the Basic Local Alignment Search Tool (BLAST)—the original paper:

- S. Altschul, W. Gish, W. Miller, E. Myers, and D. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215(3):403–410, October 5, 1990.

Sequential implementations of BLAST:

- FSA-BLAST. <http://www.fsa-blast.org/>
- NCBI BLAST. <http://www.ncbi.nlm.nih.gov/>
- WU-BLAST. <http://blast.wustl.edu/>

Parallel implementations of BLAST:

- mpiBLAST. <http://www.mpiblast.org/>
- ScalaBLAST. <http://hpc.pnl.gov/projects/scalablast/>



# A First Parallel Program

in which we build a simple sequential program; we convert it to a program for an SMP parallel computer; we see how long it takes to run each version; and we get some insight into how parallel programs execute

## 4.1 Sequential Program

To demonstrate a program that can benefit from running on a parallel computer, let's invent a simple computation that will take a long time. Here is a Java subroutine that decides whether a number  $x$  is prime using the **trial division** algorithm. The subroutine tries to divide  $x$  by 2 and by every odd number  $p$  from 3 up to the square root of  $x$ . If any remainder is 0, then  $p$  is a factor of  $x$  and  $x$  is not prime; otherwise  $x$  is prime. While trial division is by no means the fastest way to test primality, it suffices for this demonstration program.

```
private static boolean isPrime
(long x)
{
    if (x % 2 == 0) return false;
    long p = 3;
    long psqr = p*p;
    while (psqr <= x)
    {
        if (x % p == 0) return false;
        p += 2;
        psqr = p*p;
    }
    return true;
}
```

Here is a main program that uses a loop to call the subroutine with the values of  $x$  specified on the command line.

```
static int n;
static long[] x;

public static void main
(String[] args)
throws Exception
{
    n = args.length;
```

```

x = new long [n];
for (int i = 0; i < n; ++ i)
{
    x[i] = Long.parseLong (args[i]);
}
for (int i = 0; i < n; ++ i)
{
    isPrime (x[i]);
}
}

```

When we run the primality testing program, we want to know the time when each subroutine call starts and the time when each subroutine call finishes, relative to the time the program started. This tells us how long it took to run the subroutine. To measure these times, we use Java's `System.currentTimeMillis()` method, which returns the wall clock time in milliseconds (msec). We record each instant in a variable, and postpone printing the results, so as to disturb the timing as little as possible while the program is running. It can take several msec to call `println()`, and we don't want to include that time in our measurements.

```

static int n;
static long[] x;
static long t1, t2[], t3[];

public static void main
    (String[] args)
    throws Exception
    {
        t1 = System.currentTimeMillis();
        n = args.length;
        x = new long [n];
        for (int i = 0; i < n; ++ i)
        {
            x[i] = Long.parseLong (args[i]);
        }
        t2 = new long [n];
        t3 = new long [n];
        for (int i = 0; i < n; ++ i)
        {
            t2[i] = System.currentTimeMillis();
            isPrime (x[i]);
            t3[i] = System.currentTimeMillis();
        }
    }
}

```

Here is the complete Java class, Program1Seq, including code to print the running time measurements.

```
public class Program1Seq
{
    static int n;
    static long[] x;
    static long t1, t2[], t3[];

    public static void main
        (String[] args)
        throws Exception
    {
        t1 = System.currentTimeMillis();
        n = args.length;
        x = new long [n];
        for (int i = 0; i < n; ++ i)
        {
            x[i] = Long.parseLong (args[i]);
        }
        t2 = new long [n];
        t3 = new long [n];
        for (int i = 0; i < n; ++ i)
        {
            t2[i] = System.currentTimeMillis();
            isPrime (x[i]);
            t3[i] = System.currentTimeMillis();
        }
        for (int i = 0; i < n; ++ i)
        {
            System.out.println
                ("i = "+i+" call start = "+(t2[i]-t1)+" msec");
            System.out.println
                ("i = "+i+" call finish = "+(t3[i]-t1)+" msec");
        }
    }

    private static boolean isPrime
        (long x)
    {
        if (x % 2 == 0) return false;
        long p = 3;
        long psqr = p*p;
        while (psqr <= x)
```

```

    {
        if (x % p == 0) return false;
        p += 2;
        psqr = p*p;
    }
    return true;
}
}

```

## 4.2 Running the Sequential Program

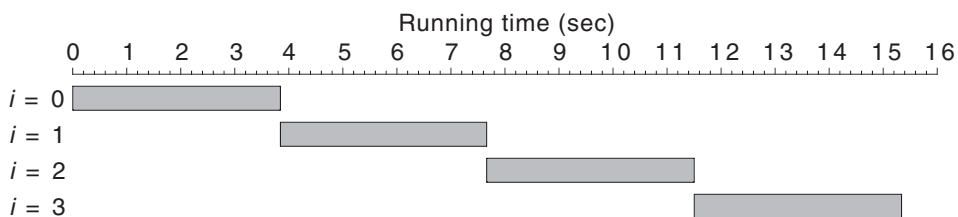
When run on an SMP parallel computer, Program1Seq prints the following. The parallel computer has four processors, each a 450 MHz Sun Microsystems UltraSPARC-II CPU, and 2 GB of shared main memory. (Running the program on a different computer would, in general, yield different results.) All four arguments happen to be prime numbers.

```

$ java Program1Seq 10000000000000037 10000000000000091 \
  100000000000000159 100000000000000187
i = 0 call start = 1 msec
i = 0 call finish = 3842 msec
i = 1 call start = 3842 msec
i = 1 call finish = 7663 msec
i = 2 call start = 7663 msec
i = 2 call finish = 11502 msec
i = 3 call start = 11502 msec
i = 3 call finish = 15342 msec

```

Plotting each subroutine call's start and finish on a timeline reveals how the program executes (Figure 4.1). The program executes each subroutine call in its entirety before going on to the next subroutine call. Because the program's statements are executed in sequence with no overlap in time, we call it a **sequential program**. There is no parallelism, even when running on a parallel computer.



**Figure 4.1** Program1Seq execution timeline, SMP parallel computer

## 4.3 SMP Parallel Program

Now let's rewrite the program using Parallel Java so it will run in parallel when executed on an SMP parallel computer. In the main program, after extracting the command line arguments, we create a **parallel team** object. The constructor argument, *n*, says we want as many threads in the parallel team as there are values to test for primality.

```
public static void main
    (String[] args)
    throws Exception
    {
        n = args.length;
        x = new long [n];
        for (int i = 0; i < n; ++ i)
            {
                x[i] = Long.parseLong (args[i]);
            }
        new ParallelTeam(n);
    }
```

Each thread in the parallel team simultaneously executes the code in a **parallel region** object, declared here as an anonymous inner class. The actual parallel code goes in the parallel region's `run()` method.

```
public static void main
    (String[] args)
    throws Exception
    {
        n = args.length;
        x = new long [n];
        for (int i = 0; i < n; ++ i)
            {
                x[i] = Long.parseLong (args[i]);
            }
        new ParallelTeam(n).execute (new ParallelRegion()
            {
                public void run()
                {
                }
            }
        ));
    }
```

Rather than use a loop to execute the computations (subroutine calls) in sequence, we want the threads to execute the computations in parallel. To make this happen, we put the code for one computation in the parallel region's `run()` method. However, we want each computation to use a different  $x$  value. To make this happen, we set `i` to the index of the calling thread within the parallel team (0 through 3, as returned by the parallel region's `getThreadIndex()` method).

```
public static void main
    (String[] args)
    throws Exception
    {
        n = args.length;
        x = new long [n];
        for (int i = 0; i < n; ++ i)
            {
                x[i] = Long.parseLong (args[i]);
            }
        new ParallelTeam(n).execute (new ParallelRegion()
            {
                public void run()
                {
                    int i = getThreadIndex();
                    isPrime (x[i]);
                }
            });
    }
}
```

Here is the complete Java class, `Program1Smp`, including code to record the running time measurements and print them after the parallel region has finished executing.

```
import edu.rit.pj.ParallelRegion;
import edu.rit.pj.ParallelTeam;
public class Program1Smp
    {
        static int n;
        static long[] x;
        static long t1, t2[], t3[];

        public static void main
            (String[] args)
            throws Exception
```

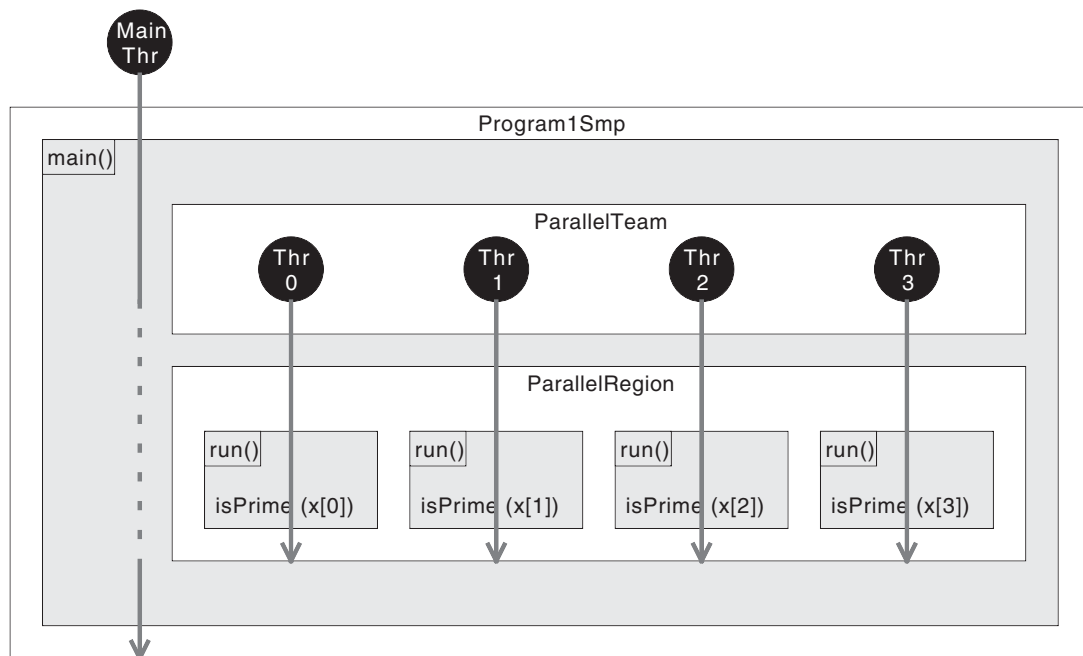
```
{
    t1 = System.currentTimeMillis();
    n = args.length;
    x = new long [n];
    for (int i = 0; i < n; ++ i)
    {

        x[i] = Long.parseLong (args[i]);
    }
    t2 = new long [n];
    t3 = new long [n];
    new ParallelTeam(n).execute (new ParallelRegion()
    {
        public void run()
        {
            int i = getThreadIndex();
            t2[i] = System.currentTimeMillis();
            isPrime (x[i]);
            t3[i] = System.currentTimeMillis();
        }
    });
    for (int i = 0; i < n; ++ i)
    {
        System.out.println
            ("i = "+i+" call start = "+(t2[i]-t1)+" msec");
        System.out.println
            ("i = "+i+" call finish = "+(t3[i]-t1)+" msec");
    }
}

private static boolean isPrime
(long x)
{
    if (x % 2 == 0) return false;
    long p = 3;
    long psqr = p*p;
    while (psqr <= x)
    {
        if (x % p == 0) return false;
        p += 2;
        psqr = p*p;
    }
    return true;
}
}
```



Here's how the program works (Figure 4.2). The main program begins with one thread, the “main thread,” executing the `main()` method. When the main thread creates the parallel team object, the parallel team object creates additional hidden threads; the constructor argument specifies the number of threads. These form a “team” of threads for executing code in parallel. When the main thread calls the parallel region's `execute()` method, the main thread suspends execution and the parallel team threads take over. All the team threads call the parallel region's `run()` method simultaneously, each thread retrieves a value for  $x$ , and each thread calls `isPrime()`. Thus, the `isPrime()` subroutine calls happen at the same time, and each subroutine call is performed by a different thread with a different argument. When all the subroutine calls have finished executing, the main thread resumes executing statements after the parallel region and prints the timing measurements.



**Figure 4.2** Program1Smp operation

When running such a thread-based program on an SMP parallel computer, the Java Virtual Machine (JVM) and the operating system are responsible for scheduling each thread to execute on a different processor. Thus, the computations done by each thread—in this case, the different subroutine calls—are executed in parallel on different processors, resulting in a speedup with respect to the sequential program.

The parallel program illustrates a central theme of parallel program design: *Repetition does not necessarily imply sequencing*. The sequential program used a loop to get  $n$  repetitions of a subroutine call. As a side effect, the loop did the repetitions *in sequence*. However, for this program there is no need to do the repetitions in sequence. We wrote the original program with a loop because Java, like many programming languages, only has constructs for expressing a *sequence* of repetitions (a loop). So accustomed are we to this feature that whenever we are confronted with a repeated calculation, we automatically think “loop.” However, a loop is not the only way to do a repeated calculation. Provided the

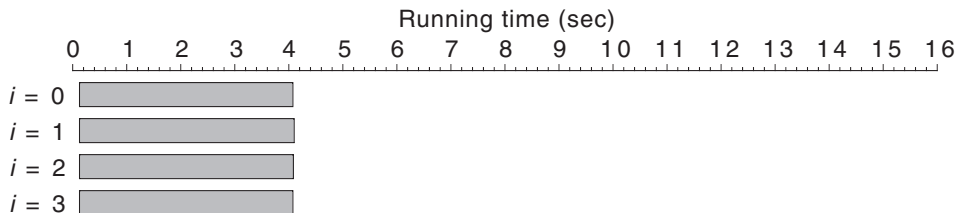
repetitions do not have to be done in sequence, another way to do a repeated calculation is to run several copies of the calculation in multiple threads. A large part of the effort in learning parallel program design is breaking the habit of always using a loop to do repetitions in sequence, and forming the new habit of doing repetitions in parallel whenever possible.

## 4.4 Running the Parallel Program

When run on the four-processor parallel computer, Program1Smp printed the following:

```
$ java Program1Smp 1000000000000037 1000000000000091 \
  10000000000000159 10000000000000187
i = 0 call start = 125 msec
i = 0 call finish = 4076 msec
i = 1 call start = 125 msec
i = 1 call finish = 4098 msec
i = 2 call start = 125 msec
i = 2 call finish = 4082 msec
i = 3 call start = 125 msec
i = 3 call finish = 4076 msec
```

Now the timeline (Figure 4.3) shows parallelism. (Compare Figures 4.1 and 4.3 to Figure 1.2.) All the computations start at the same time, execute simultaneously, and finish at about the same time. Whereas the sequential version's running time was 15342 msec, the parallel version's running time on four processors was 4098 msec—a reduction by a factor of about four, as expected.



**Figure 4.3** Program1Smp execution timeline, SMP parallel computer

To be precise, the speedup (the reduction factor) was  $15342/4098 = 3.744$ . The speedup was somewhat less than 4 because of overhead in the parallel version that is not present in the sequential version. With Program1Smp, the first subroutine call didn't begin until 125 msec after the program started. During this time, the program was occupied in creating the parallel team and parallel region objects, starting up the parallel team threads, and executing the parallel region's `run()` method—work that the sequential program didn't have to do.

This illustrates another central theme of parallel program design: *Parallelism is not free*. The benefit of speedup or sizeup comes with a price of extra overhead that is not needed in a sequential program. The name of the game is to minimize this extra overhead.

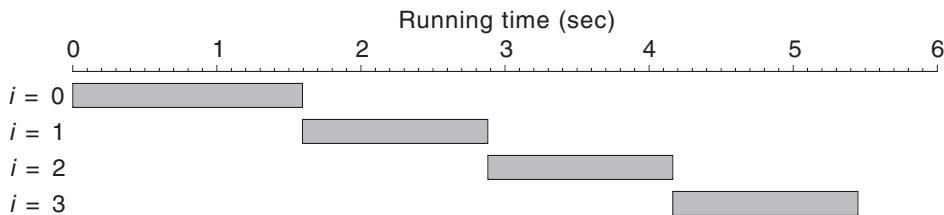
## 4.5 Running on a Regular Computer

Although intended to run on a parallel computer, Program1Seq and Program1Smp are perfectly happy to run on a nonparallel computer. In fact, one benefit of programming in Parallel Java is that you can develop and test parallel programs on any computer, and then you can shift to a parallel computer when the program is debugged and ready for usage.

Let's look at what happens when we run these programs on a regular computer. This was a non-parallel computer with a 1.6 GHz Intel Pentium CPU and 512 MB of main memory. The sequential Program1Seq program printed the following:

```
$ java Program1Seq 10000000000000037 10000000000000091 \
  100000000000000159 100000000000000187
i = 0 call start = 0 msec
i = 0 call finish = 1594 msec
i = 1 call start = 1594 msec
i = 1 call finish = 2881 msec
i = 2 call start = 2881 msec
i = 2 call finish = 4165 msec
i = 3 call start = 4165 msec
i = 3 call finish = 5450 msec
```

The timeline (Figure 4.4) shows the typical pattern of sequential execution.



**Figure 4.4** Program1Seq execution timeline, regular computer

Suppose we run the multithreaded Program1Smp program on the regular computer. Because each subroutine call will now run in a different thread, we would expect all the subroutine calls to start at roughly the same time near the beginning of the program. But because all the threads will share the same processor, and the processor will execute one thread at a time and switch to another thread every so often, we would expect the overall running time to be about the same as the sequential program. Here is what the parallel program printed.

```
$ java Program1Smp 10000000000000037 10000000000000091 \
  100000000000000159 100000000000000187
i = 0 call start = 21 msec
i = 0 call finish = 5190 msec
```

```
i = 1 call start = 71 msec
i = 1 call finish = 5053 msec
i = 2 call start = 91 msec
i = 2 call finish = 5134 msec
i = 3 call start = 14 msec
i = 3 call finish = 4981 msec
```

The timeline for this run (Figure 4.5) is about what we expected—except for one thing. The overall running time was 260 msec shorter for the four-thread parallel version than for the single-thread sequential version. We got a slight but noticeable speedup when we went from one thread to four threads on the regular computer. How can this be, when there was only one processor?

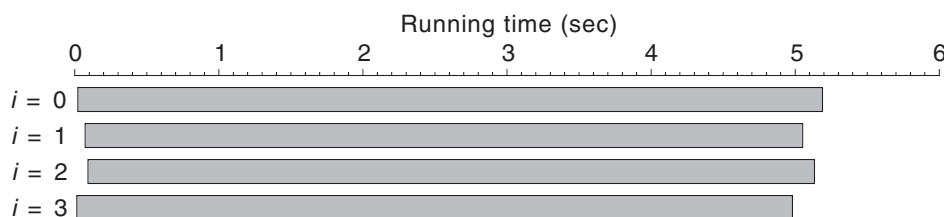


Figure 4.5 Program1Smp execution timeline, regular computer

The reason has to do with how the JVM works. A modern JVM includes a **just-in-time (JIT) compiler** that converts the Java bytecode instructions into native machine code instructions as the program runs. The JVM then executes the machine code directly instead of interpreting the Java bytecode; this greatly increases the program's execution speed. Furthermore, a modern JVM monitors which sections of bytecode are executed most frequently and compiles just those sections to machine code, leaving the remaining sections as interpreted bytecode. This avoids spending the time it would take to compile infrequently used sections of bytecode. (Sun Microsystems refers to this as a **HotSpot JVM**.) However, it takes a certain amount of execution before the JVM detects that the `isPrime()` subroutine is a hot spot and compiles it to machine code. With four threads all calling the subroutine at the same time, the JVM can detect the hot spot, and compile it to machine code, sooner in the parallel version than in the sequential version. This allows more of the parallel version's running time to be executed in the faster machine code mode, thus reducing the parallel version's running time compared to the sequential version. (To verify that this is in fact what's going on, try running both versions with the JIT compiler disabled; the parallel version then invariably takes longer than the sequential version due to the parallel version's extra overhead.) We will see further instances of how the JVM's behavior influences program performance as we study parallel programming in Java.

## 4.6 The Rest of the Book

Let's step back and look at what we've done. We started with a problem statement. We wrote a sequential program and a parallel program to solve the problem. The parallel program illustrated both general parallel programming techniques (in this case, achieving repetition via multiple threads) and specific Parallel

Java features (parallel team and parallel region). Then we ran the sequential and parallel programs, measured their running times, and gained some insight about parallel programming by comparing the programs' performance.

The rest of the book will be much the same—solving a series of problems that are chosen to illustrate various parallel programming techniques and studying the programs' performance measurements. In Part II, we will begin with SMP parallel programs, because those are quite similar to regular sequential programs. Then, in Part III, we will move on to cluster parallel programs, which are a bit more different from regular sequential programs due to the explicit message passing that is needed. In Part IV, we will combine techniques for SMP parallel programming and techniques for cluster parallel programming to write hybrid parallel programs. While the problems we solve in Parts II through IV will be interesting and perhaps fun, they were chosen solely for pedagogical reasons—to illustrate parallel programming techniques—and are not necessarily problems with any great significance in the real world. Finally, in Part V, we will apply the techniques we've learned to solve some *real-world* problems using parallel computing.

## 4.7 For Further Information

On the HotSpot JVM, and performance tuning of Java programs in general:

- Steve Wilson and Jeff Kesselman. *Java Platform Performance: Strategies and Tactics*. Addison-Wesley, 2000. Available online at:  
<http://java.sun.com/docs/books/performance/>
- Java Performance Documentation.  
<http://java.sun.com/docs/performance/index.html>