CHAPTER 1

A Pattern Language for Parallel Programming

1.1 INTRODUCTION
1.2 PARALLEL PROGRAMMING
1.3 DESIGN PATTERNS AND PATTERN LANGUAGES
1.4 A PATTERN LANGUAGE FOR PARALLEL PROGRAMMING

1.1 INTRODUCTION

Computers are used to model physical systems in many fields of science, medicine, and engineering. Modelers, whether trying to predict the weather or render a scene in the next blockbuster movie, can usually use whatever computing power is available to make ever more detailed simulations. Vast amounts of data, whether customer shopping patterns, telemetry data from space, or DNA sequences, require analysis. To deliver the required power, computer designers combine multiple processing elements into a single larger system. These so-called *parallel computers* run multiple tasks simultaneously and solve bigger problems in less time.

Traditionally, parallel computers were rare and available for only the most critical problems. Since the mid-1990s, however, the availability of parallel computers has changed dramatically. With multithreading support built into the latest microprocessors and the emergence of multiple processor cores on a single silicon die, parallel computers are becoming ubiquitous. Now, almost every university computer science department has at least one parallel computer. Virtually all oil companies, automobile manufacturers, drug development companies, and special effects studios use parallel computing.

For example, in computer animation, rendering is the step where information from the animation files, such as lighting, textures, and shading, is applied to 3D models to generate the 2D image that makes up a frame of the film. Parallel computing is essential to generate the needed number of frames (24 per second) for a feature-length film. *Toy Story,* the first completely computer-generated feature-length film, released by Pixar in 1995, was processed on a "renderfarm" consisting of 100 dual-processor machines [PS00]. By 1999, for *Toy Story 2,* Pixar was using a 1,400-processor system with the improvement in processing power fully reflected in the improved details in textures, clothing, and atmospheric effects. *Monsters, Inc.* (2001) used a system of 250 enterprise servers each containing 14 processors

for a total of 3,500 processors. It is interesting that the amount of time required to generate a frame has remained relatively constant—as computing power (both the number of processors and the speed of each processor) has increased, it has been exploited to improve the quality of the animation.

The biological sciences have taken dramatic leaps forward with the availability of DNA sequence information from a variety of organisms, including humans. One approach to sequencing, championed and used with success by Celera Corp., is called the whole genome shotgun algorithm. The idea is to break the genome into small segments, experimentally determine the DNA sequences of the segments, and then use a computer to construct the entire sequence from the segments by finding overlapping areas. The computing facilities used by Celera to sequence the human genome included 150 four-way servers plus a server with 16 processors and 64GB of memory. The calculation involved 500 million trillion base-to-base comparisons [Ein00].

The SETI@home project [SET, ACK+02] provides a fascinating example of the power of parallel computing. The project seeks evidence of extraterrestrial intelligence by scanning the sky with the world's largest radio telescope, the Arecibo Telescope in Puerto Rico. The collected data is then analyzed for candidate signals that might indicate an intelligent source. The computational task is beyond even the largest supercomputer, and certainly beyond the capabilities of the facilities available to the SETI@home project. The problem is solved with *public resource computing,* which turns PCs around the world into a huge parallel computer connected by the Internet. Data is broken up into work units and distributed over the Internet to client computers whose owners donate spare computing time to support the project. Each client periodically connects with the SETI@home server, downloads the data to analyze, and then sends the results back to the server. The client program is typically implemented as a screen saver so that it will devote CPU cycles to the SETI problem only when the computer is otherwise idle. A work unit currently requires an average of between seven and eight hours of CPU time on a client. More than 205,000,000 work units have been processed since the start of the project. More recently, similar technology to that demonstrated by SETI@home has been used for a variety of public resource computing projects as well as internal projects within large companies utilizing their idle PCs to solve problems ranging from drug screening to chip design validation.

Although computing in less time is beneficial, and may enable problems to be solved that couldn't be otherwise, it comes at a cost. Writing software to run on parallel computers can be difficult. Only a small minority of programmers have experience with parallel programming. If all these computers designed to exploit parallelism are going to achieve their potential, more programmers need to learn how to write parallel programs.

This book addresses this need by showing competent programmers of sequential machines how to design programs that can run on parallel computers. Although many excellent books show how to use particular parallel programming environments, this book is unique in that it focuses on how to think about and design parallel algorithms. To accomplish this goal, we will be using the concept of a *pattern language.* This highly structured representation of expert design experience has been heavily used in the object-oriented design community.

The book opens with two introductory chapters. The first gives an overview of the parallel computing landscape and background needed to understand and use the pattern language. This is followed by a more detailed chapter in which we lay out the basic concepts and jargon used by parallel programmers. The book then moves into the pattern language itself.

1.2 PARALLEL PROGRAMMING

The key to parallel computing is *exploitable concurrency.* Concurrency exists in a computational problem when the problem can be decomposed into subproblems that can safely execute at the same time. To be of any use, however, it must be possible to structure the code to expose and later exploit the concurrency and permit the subproblems to actually run concurrently; that is, the concurrency must be *exploitable.*

Most large computational problems contain exploitable concurrency. A programmer works with exploitable concurrency by creating a parallel algorithm and implementing the algorithm using a parallel programming environment. When the resulting parallel program is run on a system with multiple processors, the amount of time we have to wait for the results of the computation is reduced. In addition, multiple processors may allow larger problems to be solved than could be done on a single-processor system.

As a simple example, suppose part of a computation involves computing the summation of a large set of values. If multiple processors are available, instead of adding the values together sequentially, the set can be partitioned and the summations of the subsets computed simultaneously, each on a different processor. The partial sums are then combined to get the final answer. Thus, using multiple processors to compute in parallel may allow us to obtain a solution sooner. Also, if each processor has its own memory, partitioning the data between the processors may allow larger problems to be handled than could be handled on a single processor.

This simple example shows the essence of parallel computing. The goal is to use multiple processors to solve problems in less time and/or to solve bigger problems than would be possible on a single processor. The programmer's task is to identify the concurrency in the problem, structure the algorithm so that this concurrency can be exploited, and then implement the solution using a suitable programming environment. The final step is to solve the problem by executing the code on a parallel system.

Parallel programming presents unique challenges. Often, the concurrent tasks making up the problem include dependencies that must be identified and correctly managed. The order in which the tasks execute may change the answers of the computations in nondeterministic ways. For example, in the parallel summation described earlier, a partial sum cannot be combined with others until its own computation has completed. The algorithm imposes a partial order on the tasks (that is, they must complete before the sums can be combined). More subtly, the numerical value of the summations may change slightly depending on the order of the operations within the sums because floating-point arithmetic is nonassociative. A good parallel programmer must take care to ensure that nondeterministic issues

such as these do not affect the quality of the final answer. Creating safe parallel programs can take considerable effort from the programmer.

Even when a parallel program is "correct", it may fail to deliver the anticipated performance improvement from exploiting concurrency. Care must be taken to ensure that the overhead incurred by managing the concurrency does not overwhelm the program runtime. Also, partitioning the work among the processors in a balanced way is often not as easy as the summation example suggests. The effectiveness of a parallel algorithm depends on how well it maps onto the underlying parallel computer, so a parallel algorithm could be very effective on one parallel architecture and a disaster on another.

We will revisit these issues and provide a more quantitative view of parallel computation in the next chapter.

1.3 DESIGN PATTERNS AND PATTERN LANGUAGES

A *design pattern* describes a good solution to a recurring problem in a particular context. The pattern follows a prescribed format that includes the pattern name, a description of the context, the forces (goals and constraints), and the solution. The idea is to record the experience of experts in a way that can be used by others facing a similar problem. In addition to the solution itself, the name of the pattern is important and can form the basis for a domain-specific vocabulary that can significantly enhance communication between designers in the same area.

Design patterns were first proposed by Christopher Alexander. The domain was city planning and architecture [AIS77]. Design patterns were originally introduced to the software engineering community by Beck and Cunningham [BC87] and became prominent in the area of object-oriented programming with the publication of the book by Gamma, Helm, Johnson, and Vlissides [GHJV95], affectionately known as the GoF (Gang of Four) book. This book gives a large collection of design patterns for object-oriented programming. To give one example, the *Visitor pattern* describes a way to structure classes so that the code implementing a heterogeneous data structure can be kept separate from the code to traverse it. Thus, what happens in a traversal depends on both the type of each node and the class that implements the traversal. This allows multiple functionality for data structure traversals, and significant flexibility as new functionality can be added without having to change the data structure class. The patterns in the GoF book have entered the lexicon of object-oriented programming—references to its patterns are found in the academic literature, trade publications, and system documentation. These patterns have by now become part of the expected knowledge of any competent software engineer.

An educational nonprofit organization called the Hillside Group [Hil] was formed in 1993 to promote the use of patterns and pattern languages and, more generally, to improve human communication about computers "by encouraging people to codify common programming and design practice". To develop new patterns and help pattern writers hone their skills, the Hillside Group sponsors an annual Pattern Languages of Programs (PLoP) workshop and several spinoffs in other parts of the world, such as ChiliPLoP (in the western United States), KoalaPLoP (Australia), EuroPLoP (Europe), and Mensore PLoP (Japan). The proceedings of these workshops [Pat] provide a rich source of patterns covering a vast range of application domains in software development and have been used as a basis for several books [CS95, VCK96, MRB97, HFR99].

In his original work on patterns, Alexander provided not only a catalog of patterns, but also a *pattern language* that introduced a new approach to design. In a pattern language, the patterns are organized into a structure that leads the user through the collection of patterns in such a way that complex systems can be designed using the patterns. At each decision point, the designer selects an appropriate pattern. Each pattern leads to other patterns, resulting in a final design in terms of a web of patterns. Thus, a pattern language embodies a design methodology and provides domain-specific advice to the application designer. (In spite of the overlapping terminology, a pattern language is *not* a programming language.)

1.4 A PATTERN LANGUAGE FOR PARALLEL PROGRAMMING

This book describes a pattern language for parallel programming that provides several benefits. The immediate benefits are a way to disseminate the experience of experts by providing a catalog of good solutions to important problems, an expanded vocabulary, and a methodology for the design of parallel programs. We hope to lower the barrier to parallel programming by providing guidance through the entire process of developing a parallel program. The programmer brings to the process a good understanding of the actual problem to be solved and then works through the pattern language, eventually obtaining a detailed parallel design or possibly working code. In the longer term, we hope that this pattern language can provide a basis for both a disciplined approach to the qualitative evaluation of different programming models and the development of parallel programming tools.

The pattern language is organized into four design spaces—*Finding Concurrency, Algorithm Structure, Supporting Structures,* and *Implementation Mechanisms*—which form a linear hierarchy, with *Finding Concurrency* at the top and *Implementation Mechanisms* at the bottom, as shown in Fig. 1.1.

The *Finding Concurrency* design space is concerned with structuring the problem to expose exploitable concurrency. The designer working at this level focuses

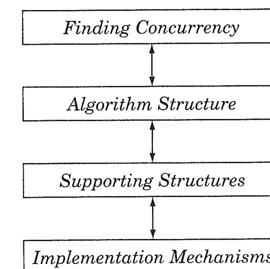

Figure 1.1: Overview of the pattern language

on high-level algorithmic issues and reasons about the problem to expose potential concurrency. The *Algorithm Structure* design space is concerned with structuring the algorithm to take advantage of potential concurrency. That is, the designer working at this level reasons about how to use the concurrency exposed in working with the *Finding Concurrency* patterns. The *Algorithm Structure* patterns describe overall strategies for exploiting concurrency. The *Supporting Structures* design space represents an intermediate stage between the *Algorithm Structure* and *Implementation Mechanisms* design spaces. Two important groups of patterns in this space are those that represent program-structuring approaches and those that represent commonly used shared data structures. The *Implementation Mechanisms* design space is concerned with how the patterns of the higher-level spaces are mapped into particular programming environments. We use it to provide descriptions of common mechanisms for process/thread management (for example, creating or destroying processes/threads) and process/thread interaction (for example, semaphores, barriers, or message passing). The items in this design space are not presented as patterns because in many cases they map directly onto elements within particular parallel programming environments. They are included in the pattern language anyway, however, to provide a complete path from problem description to code.

CHAPTER 2

Background and Jargon of Parallel Computing

2.1 CONCURRENCY IN PARALLEL PROGRAMS VERSUS OPERATING SYSTEMS
2.2 PARALLEL ARCHITECTURES: A BRIEF INTRODUCTION
2.3 PARALLEL PROGRAMMING ENVIRONMENTS
2.4 THE JARGON OF PARALLEL COMPUTING
2.5 A QUANTITATIVE LOOK AT PARALLEL COMPUTATION
2.6 COMMUNICATION
2.7 SUMMARY

In this chapter, we give an overview of the parallel programming landscape, and define any specialized parallel computing terminology that we will use in the patterns. Because many terms in computing are overloaded, taking different meanings in different contexts, we suggest that even readers familiar with parallel programming at least skim this chapter.

2.1 CONCURRENCY IN PARALLEL PROGRAMS VERSUS OPERATING SYSTEMS

Concurrency was first exploited in computing to better utilize or share resources within a computer. Modern operating systems support context switching to allow multiple tasks to appear to execute concurrently, thereby allowing useful work to occur while the processor is stalled on one task. This application of concurrency, for example, allows the processor to stay busy by swapping in a new task to execute while another task is waiting for I/O. By quickly swapping tasks in and out, giving each task a "slice" of the processor time, the operating system can allow multiple users to use the system as if each were using it alone (but with degraded performance).

Most modern operating systems can use multiple processors to increase the throughput of the system. The UNIX shell uses concurrency along with a communication abstraction known as *pipes* to provide a powerful form of modularity: Commands are written to accept a stream of bytes as input (the *consumer*) and produce a stream of bytes as output (the *producer*). Multiple commands can be chained together with a pipe connecting the output of one command to the input of the next, allowing complex commands to be built from simple building blocks. Each command is executed in its own process, with all processes executing concurrently. Because the producer blocks if buffer space in the pipe is not available, and the consumer blocks if data is not available, the job of managing the stream of results moving between commands is greatly simplified. More recently, with

operating systems with windows that invite users to do more than one thing at a time, and the Internet, which often introduces I/O delays perceptible to the user, almost every program that contains a GUI incorporates concurrency.

Although the fundamental concepts for safely handling concurrency are the same in parallel programs and operating systems, there are some important differences. For an operating system, the problem is not finding concurrency—the concurrency is inherent in the way the operating system functions in managing a collection of concurrently executing processes (representing users, applications, and background activities such as print spooling) and providing synchronization mechanisms so resources can be safely shared. However, an operating system must support concurrency in a robust and secure way: Processes should not be able to interfere with each other (intentionally or not), and the entire system should not crash if something goes wrong with one process. In a parallel program, finding and exploiting concurrency can be a challenge, while isolating processes from each other is not the critical concern it is with an operating system. Performance goals are different as well. In an operating system, performance goals are normally related to throughput or response time, and it may be acceptable to sacrifice some efficiency to maintain robustness and fairness in resource allocation. In a parallel program, the goal is to minimize the running time of a single program.

2.2 PARALLEL ARCHITECTURES: A BRIEF INTRODUCTION

There are dozens of different parallel architectures, among them networks of workstations, clusters of off-the-shelf PCs, massively parallel supercomputers, tightly coupled symmetric multiprocessors, and multiprocessor workstations. In this section, we give an overview of these systems, focusing on the characteristics relevant to the programmer.

2.2.1 Flynn's Taxonomy

By far the most common way to characterize these architectures is Flynn's taxonomy [Fly72]. He categorizes all computers according to the number of instruction streams and data streams they have, where a stream is a sequence of instructions or data on which a computer operates. In Flynn's taxonomy, there are four possibilities: SISD, SIMD, MISD, and MIMD.

Single Instruction, Single Data (SISD). In a SISD system, one stream of instructions processes a single stream of data, as shown in Fig. 2.1. This is the common von Neumann model used in virtually all single-processor computers.

Single Instruction, Multiple Data (SIMD). In a SIMD system, a single instruction stream is concurrently broadcast to multiple processors, each with its own data stream (as shown in Fig. 2.2). The original systems from Thinking Machines and MasPar can be classified as SIMD. The CPP DAP Gamma II and Quadrics Apemille are more recent examples; these are typically deployed in specialized applications, such as digital signal processing, that are suited to fine-grained parallelism and require little interprocess communication. Vector processors, which

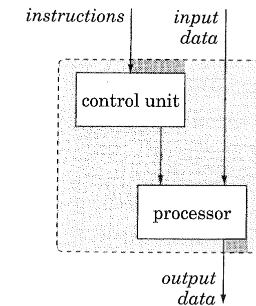

Figure 2.1: The Single Instruction, Single Data (SISD) architecture

operate on vector data in a pipelined fashion, can also be categorized as SIMD. Exploiting this parallelism is usually done by the compiler.

Multiple Instruction, Single Data (MISD). No well-known systems fit this designation. It is mentioned for the sake of completeness.

Multiple Instruction, Multiple Data (MIMD). In a MIMD system, each processing element has its own stream of instructions operating on its own data. This architecture, shown in Fig. 2.3, is the most general of the architectures in that each of the other cases can be mapped onto the MIMD architecture. The vast majority of modern parallel systems fit into this category.

2.2.2 A Further Breakdown of MIMD

The MIMD category of Flynn's taxonomy is too broad to be useful on its own; this category is typically decomposed according to memory organization.

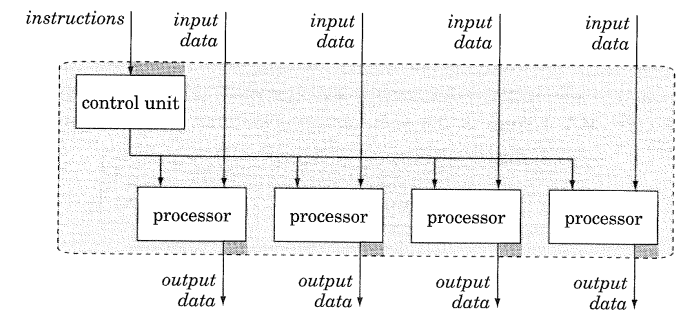

Figure 2.2: The Single Instruction, Multiple Data (SIMD) architecture

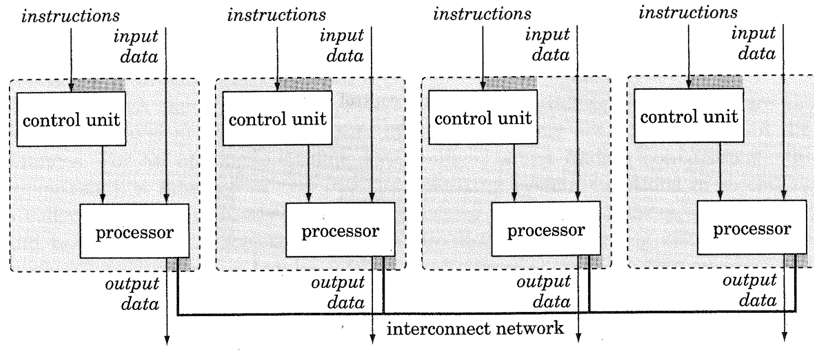

Figure 2.3: The Multiple Instruction, Multiple Data (MIMD) architecture

Shared memory. In a shared-memory system, all processes share a single address space and communicate with each other by writing and reading shared variables.

One class of shared-memory systems is called SMPs (symmetric multiprocessors). As shown in Fig. 2.4, all processors share a connection to a common memory and access all memory locations at equal speeds. SMP systems are arguably the easiest parallel systems to program because programmers do not need to distribute data structures among processors. Because increasing the number of processors increases contention for the memory, the processor/memory bandwidth is typically a limiting factor. Thus, SMP systems do not scale well and are limited to small numbers of processors.

The other main class of shared-memory systems is called NUMA (nonuniform memory access). As shown in Fig. 2.5, the memory is shared; that is, it is uniformly addressable from all processors, but some blocks of memory may be physically more closely associated with some processors than others. This reduces the memory bandwidth bottleneck and allows systems with more processors; however, as a result, the access time from a processor to a memory location can be significantly different depending on how "close" the memory location is to the processor. To mitigate the effects of nonuniform access, each processor has a cache, along with a protocol to keep cache entries coherent. Hence, another name for these architectures is cache-coherent nonuniform memory access systems (ccNUMA). Logically, programming a ccNUMA system is the same as programming an SMP, but to obtain the best performance, the programmer will need to be more careful about locality issues and cache effects.

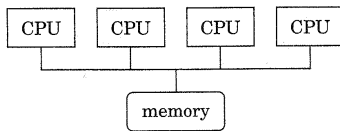

Figure 2.4: The Symmetric Multiprocessor (SMP) architecture

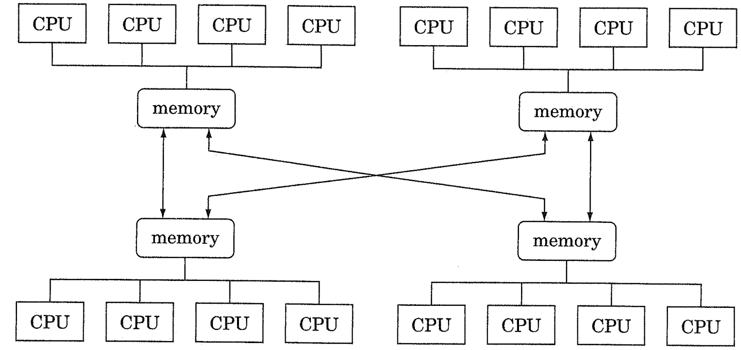

Figure 2.5: An example of the nonuniform memory access (NUMA) architecture

Distributed memory. In a distributed-memory system, each process has its own address space and communicates with other processes by *message passing* (sending and receiving messages). A schematic representation of a distributed memory computer is shown in Fig. 2.6.

Depending on the topology and technology used for the processor interconnection, communication speed can range from almost as fast as shared memory (in tightly integrated supercomputers) to orders of magnitude slower (for example, in a cluster of PCs interconnected with an Ethernet network). The programmer must explicitly program all the communication between processors and be concerned with the distribution of data.

Distributed-memory computers are traditionally divided into two classes: MPP (massively parallel processors) and clusters. In an MPP, the processors and the network infrastructure are tightly coupled and specialized for use in a parallel computer. These systems are extremely scalable, in some cases supporting the use of many thousands of processors in a single system [MSW96, IBM02].

Clusters are distributed-memory systems composed of off-the-shelf computers connected by an off-the-shelf network. When the computers are PCs running the Linux operating system, these clusters are called *Beowulf clusters*. As

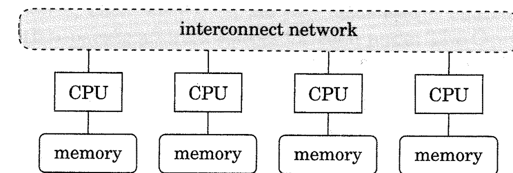

Figure 2.6: The distributed-memory architecture

off-the-shelf networking technology improves, systems of this type are becoming more common and much more powerful. Clusters provide an inexpensive way for an organization to obtain parallel computing capabilities [Beo]. Preconfigured clusters are now available from many vendors. One frugal group even reported constructing a useful parallel system by using a cluster to harness the combined power of obsolete PCs that otherwise would have been discarded [HHS01].

Hybrid systems. These systems are clusters of nodes with separate address spaces in which each node contains several processors that share memory.

According to van der Steen and Dongarra's "Overview of Recent Supercomputers" [vdSD03], which contains a brief description of the supercomputers currently or soon to be commercially available, hybrid systems formed from clusters of SMPs connected by a fast network are currently the dominant trend in high-performance computing. For example, in late 2003, four of the five fastest computers in the world were hybrid systems [Top].

Grids. Grids are systems that use distributed, heterogeneous resources connected by LANs and/or WANs [FK03]. Often the interconnection network is the Internet. Grids were originally envisioned as a way to link multiple supercomputers to enable larger problems to be solved, and thus could be viewed as a special type of distributed-memory or hybrid MIMD machine. More recently, the idea of grid computing has evolved into a general way to share heterogeneous resources, such as computation servers, storage, application servers, information services, or even scientific instruments. Grids differ from clusters in that the various resources in the grid need not have a common point of administration. In most cases, the resources on a grid are owned by different organizations that maintain control over the policies governing use of the resources. This affects the way these systems are used, the middleware created to manage them, and most importantly for this discussion, the overhead incurred when communicating between resources within the grid.

2.2.3 Summary

We have classified these systems according to the characteristics of the hardware. These characteristics typically influence the native programming model used to express concurrency on a system; however, this is not always the case. It is possible for a programming environment for a shared-memory machine to provide the programmer with the abstraction of distributed memory and message passing. Virtual distributed shared memory systems contain middleware to provide the opposite: the abstraction of shared memory on a distributed-memory machine.

2.3 PARALLEL PROGRAMMING ENVIRONMENTS

Parallel programming environments provide the basic tools, language features, and *application programming interfaces* (APIs) needed to construct a parallel program. A programming environment implies a particular abstraction of the computer system called a *programming model*. Traditional sequential computers use the well known von Neumann model. Because all sequential computers use this model,

software designers can design software to a single abstraction and reasonably expect it to map onto most, if not all, sequential computers.

Unfortunately, there are many possible models for parallel computing, reflecting the different ways processors can be interconnected to construct a parallel system. The most common models are based on one of the widely deployed parallel architectures: shared memory, distributed memory with message passing, or a hybrid combination of the two.

Programming models too closely aligned to a particular parallel system lead to programs that are not portable between parallel computers. Because the effective lifespan of software is longer than that of hardware, many organizations have more than one type of parallel computer, and most programmers insist on programming environments that allow them to write portable parallel programs. Also, explicitly managing large numbers of resources in a parallel computer is difficult, suggesting that higher-level abstractions of the parallel computer might be useful. The result is that as of the mid-1990s, there was a veritable glut of parallel programming environments. A partial list of these is shown in Table 2.1. This created a great deal of confusion for application developers and hindered the adoption of parallel computing for mainstream applications.

Fortunately, by the late 1990s, the parallel programming community converged predominantly on two environments for parallel programming: OpenMP [OMP] for shared memory and MPI [Mesb] for message passing.

OpenMP is a set of language extensions implemented as compiler directives. Implementations are currently available for Fortran, C, and C++. OpenMP is frequently used to incrementally add parallelism to sequential code. By adding a compiler directive around a loop, for example, the compiler can be instructed to generate code to execute the iterations of the loop in parallel. The compiler takes care of most of the details of thread creation and management. OpenMP programs tend to work very well on SMPs, but because its underlying programming model does not include a notion of nonuniform memory access times, it is less ideal for ccNUMA and distributed-memory machines.

MPI is a set of library routines that provide for process management, message passing, and some collective communication operations (these are operations that involve all the processes involved in a program, such as barrier, broadcast, and reduction). MPI programs can be difficult to write because the programmer is responsible for data distribution and explicit interprocess communication using messages. Because the programming model assumes distributed memory, MPI is a good choice for MPPs and other distributed-memory machines.

Neither OpenMP nor MPI is an ideal fit for hybrid architectures that combine multiprocessor nodes, each with multiple processes and a shared memory, into a larger system with separate address spaces for each node: The OpenMP model does not recognize nonuniform memory access times, so its data allocation can lead to poor performance on machines that are not SMPs, while MPI does not include constructs to manage data structures residing in a shared memory. One solution is a hybrid model in which OpenMP is used on each shared-memory node and MPI is used between the nodes. This works well, but it requires the programmer to work with two different programming models within a single program. Another option

Table 2.1: Some Parallel Programming Environments from the Mid-1990s

"C* in C	CUMULVS	Java RMI	P-RIO	Quake
ABCPL	DAGGER	javaPG	P3L	Quark
ACE	DAPPLE	JAVAR	P4-Linda	Quick Threads
ACT++	Data Parallel C	JavaSpaces	Pablo	Sage++
ADDAP	DC++	JIDL	PADE	SAM
Adl	DCE++	Joyce	PADRE	SCANDAL
Adsmith	DDD	Karma	Panda	SCHEDULE
AFAPI	DICE	Khoros	Papers	SciTL
ALWAN	DIPC	KOAN/Fortran-S	Para++	SDDA
AM	Distributed Smalltalk	LAM	Paradigm	SHMEM
AMDC	DOLIB	Legion	Parafrase2	SIMPLE
Amoeba	DOME	Lilac	Paralation	Sina
AppLeS	DOSMOS	Linda	Parallaxis	SISAL
ARTS	DRL	LiPS	Parallel Haskell	SMI
Athapascan-0b	DSM-Threads	Locust	Parallel-C++	SONiC
Aurora	Ease	Lparx	ParC	Split-C
Automap	ECO	Lucid	ParLib++	SR
bb_threads	Eilean	Maisie	ParLin	Sthreads
Blaze	Emerald	Manifold	Parlog	Strand
BlockComm	EPL	Mentat	Parmacs	SUIF
BSP	Excalibur	Meta Chaos	Parti	SuperPascal
C*	Express	Midway	pC	Synergy
C**	Falcon	Millipede	pC++	TCGMSG
C4	Filaments	Mirage	PCN	Telegraphos
CarlOS	FLASH	Modula-2*	PCP:	The FORCE
Cashmere	FM	Modula-P	PCU	Threads.h++
CC++	Fork	MOSIX	PEACE	TRAPPER
Charlotte	Fortran-M	MpC	PENNY	TreadMarks
Charm	FX	MPC++	PET	UC
Charm++	GA	MPI	PETSc	uC++
Chu	GAMMA	Multipol	PH	UNITY
Cid	Glenda	Munin	Phosphorus	V
Cilk	GLU	Nano-Threads	POET	ViC*
CM-Fortran	GUARD	NESL	Polaris	Visifold V-NUS
Code	HAsL	NetClasses++	POOL-T	VPE
Concurrent ML	HORUS	Nexus	POOMA	Win32 threads
Converse	HPC	Nimrod	POSYBL	WinPar
COOL	HPC++	NOW	PRESTO	WWWinda
CORRELATE	HPF	Objective Linda	Prospero	XENOOPS
CparPar	IMPACT	Occam	Proteus	XPC
CPS	ISETL-Linda	Omega	PSDM	Zounds
CRL	ISIS	OOF90	PSI	ZPL
CSP	JADA	Orca	PVM	
Cthreads	JADE	P++	QPC++	

is to use MPI on both the shared-memory and distributed-memory portions of the algorithm and give up the advantages of a shared-memory programming model, even when the hardware directly supports it.

New high-level programming environments that simplify portable parallel programming and more accurately reflect the underlying parallel architectures are topics of current research [Cen]. Another approach more popular in the commercial

sector is to extend MPI and OpenMP. In the mid-1990s, the MPI Forum defined an extended MPI called MPI 2.0, although implementations are not widely available at the time this was written. It is a large complex extension to MPI that includes dynamic process creation, parallel I/O, and many other features. Of particular interest to programmers of modern hybrid architectures is the inclusion of one-sided communication. One-sided communication mimics some of the features of a shared-memory system by letting one process write into or read from the memory regions of other processes. The term "one-sided" refers to the fact that the read or write is launched by the initiating process without the explicit involvement of the other participating process. A more sophisticated abstraction of one-sided communication is available as part of the Global Arrays [NHL96, NHK+02, Gloa] package. Global Arrays works together with MPI to help a programmer manage distributed array data. After the programmer defines the array and how it is laid out in memory, the program executes "puts" or "gets" into the array without needing to explicitly manage which MPI process "owns" the particular section of the array. In essence, the global array provides an abstraction of a globally shared array. This only works for arrays, but these are such common data structures in parallel computing that this package, although limited, can be very useful.

Just as MPI has been extended to mimic some of the benefits of a shared-memory environment, OpenMP has been extended to run in distributed-memory environments. The annual WOMPAT (Workshop on OpenMP Applications and Tools) workshops contain many papers discussing various approaches and experiences with OpenMP in clusters and ccNUMA environments.

MPI is implemented as a library of routines to be called from programs written in a sequential programming language, whereas OpenMP is a set of extensions to sequential programming languages. They represent two of the possible categories of parallel programming environments (libraries and language extensions), and these two particular environments account for the overwhelming majority of parallel computing being done today. There is, however, one more category of parallel programming environments, namely languages with built-in features to support parallel programming. Java is such a language. Rather than being designed to support high-performance computing, Java is an object-oriented, general-purpose programming environment with features for explicitly specifying concurrent processing with shared memory. In addition, the standard I/O and network packages provide classes that make it easy for Java to perform interprocess communication between machines, thus making it possible to write programs based on both the shared-memory and the distributed-memory models. The newer `java.nio` packages support I/O in a way that is less convenient for the programmer, but gives significantly better performance, and Java 2 1.5 includes new support for concurrent programming, most significantly in the `java.util.concurrent.*` packages. Additional packages that support different approaches to parallel computing are widely available.

Although there have been other general-purpose languages, both prior to Java and more recent (for example, C#), that contained constructs for specifying concurrency, Java is the first to become widely used. As a result, it may be the first exposure for many programmers to concurrent and parallel programming. Although

Java provides software engineering benefits, currently the performance of parallel Java programs cannot compete with OpenMP or MPI programs for typical scientific computing applications. The Java design has also been criticized for several deficiencies that matter in this domain (for example, a floating-point model that emphasizes portability and more-reproducible results over exploiting the available floating-point hardware to the fullest, inefficient handling of arrays, and lack of a lightweight mechanism to handle complex numbers). The performance difference between Java and other alternatives can be expected to decrease, especially for symbolic or other nonnumeric problems, as compiler technology for Java improves and as new packages and language extensions become available. The Titanium project [Tita] is an example of a Java dialect designed for high-performance computing in a ccNUMA environment.

For the purposes of this book, we have chosen OpenMP, MPI, and Java as the three environments we will use in our examples—OpenMP and MPI for their popularity and Java because it is likely to be many programmers' first exposure to concurrent programming. A brief overview of each can be found in the appendixes.

2.4 THE JARGON OF PARALLEL COMPUTING

In this section, we define some terms that are frequently used throughout the pattern language. Additional definitions can be found in the glossary.

Task. The first step in designing a parallel program is to break the problem up into tasks. A task is a sequence of instructions that operate together as a group. This group corresponds to some logical part of an algorithm or program. For example, consider the multiplication of two order-N matrices. Depending on how we construct the algorithm, the tasks could be (1) the multiplication of subblocks of the matrices, (2) inner products between rows and columns of the matrices, or (3) individual iterations of the loops involved in the matrix multiplication. These are all legitimate ways to define tasks for matrix multiplication; that is, the task definition follows from the way the algorithm designer thinks about the problem.

Unit of execution (UE). To be executed, a task needs to be mapped to a UE such as a process or thread. A *process* is a collection of resources that enables the execution of program instructions. These resources can include virtual memory, I/O descriptors, a runtime stack, signal handlers, user and group IDs, and access control tokens. A more high-level view is that a process is a "heavyweight" unit of execution with its own address space. A *thread* is the fundamental UE in modern operating systems. A thread is associated with a process and shares the process's environment. This makes threads lightweight (that is, a context switch between threads takes only a small amount of time). A more high-level view is that a thread is a "lightweight" UE that shares an address space with other threads.

We will use *unit of execution* or UE as a generic term for one of a collection of possibly concurrently executing entities, usually either processes or threads. This is convenient in the early stages of program design when the distinctions between processes and threads are less important.

Processing element (PE). We use the term processing element (PE) as a generic term for a hardware element that executes a stream of instructions. The unit of hardware considered to be a PE depends on the context. For example, some programming environments view each workstation in a cluster of SMP workstations as executing a single instruction stream; in this situation, the PE would be the workstation. A different programming environment running on the same hardware, however, might view each processor of each workstation as executing an individual instruction stream; in this case, the PE is the individual processor, and each workstation contains several PEs.

Load balance and load balancing. To execute a parallel program, the tasks must be mapped to UEs, and the UEs to PEs. How the mappings are done can have a significant impact on the overall performance of a parallel algorithm. It is crucial to avoid the situation in which a subset of the PEs is doing most of the work while others are idle. *Load balance* refers to how well the work is distributed among PEs. *Load balancing* is the process of allocating work to PEs, either statically or dynamically, so that the work is distributed as evenly as possible.

Synchronization. In a parallel program, due to the nondeterminism of task scheduling and other factors, events in the computation might not always occur in the same order. For example, in one run, a task might read variable x before another task reads variable y; in the next run with the same input, the events might occur in the opposite order. In many cases, the order in which two events occur does not matter. In other situations, the order does matter, and to ensure that the program is correct, the programmer must introduce synchronization to enforce the necessary ordering constraints. The primitives provided for this purpose in our selected environments are discussed in the *Implementation Mechanisms* design space (Sec. 6.3).

Synchronous versus asynchronous. We use these two terms to qualitatively refer to how tightly coupled in time two events are. If two events must happen at the same time, they are synchronous; otherwise they are asynchronous. For example, message passing (that is, communication between UEs by sending and receiving messages) is synchronous if a message sent must be received before the sender can continue. Message passing is asynchronous if the sender can continue its computation regardless of what happens at the receiver, or if the receiver can continue computations while waiting for a receive to complete.

Race conditions. A *race condition* is a kind of error peculiar to parallel programs. It occurs when the outcome of a program changes as the relative scheduling of UEs varies. Because the operating system and not the programmer controls the scheduling of the UEs, race conditions result in programs that potentially give different answers even when run on the same system with the same data. Race conditions are particularly difficult errors to debug because by their nature they cannot be reliably reproduced. Testing helps, but is not as effective as with sequential programs: A program may run correctly the first thousand times and

then fail catastrophically on the thousand-and-first execution—and then run again correctly when the programmer attempts to reproduce the error as the first step in debugging.

Race conditions result from errors in synchronization. If multiple UEs read and write shared variables, the programmer must protect access to these shared variables so the reads and writes occur in a valid order regardless of how the tasks are interleaved. When many variables are shared or when they are accessed through multiple levels of indirection, verifying by inspection that no race conditions exist can be very difficult. Tools are available that help detect and fix race conditions, such as ThreadChecker from Intel Corporation, and the problem remains an area of active and important research [NM92].

Deadlocks. Deadlocks are another type of error peculiar to parallel programs. A deadlock occurs when there is a cycle of tasks in which each task is blocked waiting for another to proceed. Because all are waiting for another task to do something, they will all be blocked forever. As a simple example, consider two tasks in a message-passing environment. Task A attempts to receive a message from task B, after which A will reply by sending a message of its own to task B. Meanwhile, task B attempts to receive a message from task A, after which B will send a message to A. Because each task is waiting for the other to send it a message first, both tasks will be blocked forever. Fortunately, deadlocks are not difficult to discover, as the tasks will stop at the point of the deadlock.

2.5 A QUANTITATIVE LOOK AT PARALLEL COMPUTATION

The two main reasons for implementing a parallel program are to obtain better performance and to solve larger problems. Performance can be both modeled and measured, so in this section we will take a another look at parallel computations by giving some simple analytical models that illustrate some of the factors that influence the performance of a parallel program.

Consider a computation consisting of three parts: a setup section, a computation section, and a finalization section. The total running time of this program on one PE is then given as the sum of the times for the three parts.

$$T_{total}(1) = T_{setup} + T_{compute} + T_{finalization} \quad (2.1)$$

What happens when we run this computation on a parallel computer with multiple PEs? Suppose that the setup and finalization sections cannot be carried out concurrently with any other activities, but that the computation section could be divided into tasks that would run independently on as many PEs as are available, with the same total number of computation steps as in the original computation. The time for the full computation on P PEs can therefore be given by

$$T_{total}(P) = T_{setup} + \frac{T_{compute}(1)}{P} + T_{finalization} \quad (2.2)$$

Of course, Eq. 2.2 describes a very idealized situation. However, the idea that computations have a serial part (for which additional PEs are useless) and a parallelizable part (for which more PEs decrease the running time) is realistic. Thus, this simple model captures an important relationship.

An important measure of how much additional PEs help is the *relative speedup* S, which describes how much faster a problem runs in a way that normalizes away the actual running time.

$$S(P) = \frac{T_{total}(1)}{T_{total}(P)} \quad (2.3)$$

A related measure is the efficiency E, which is the speedup normalized by the number of PEs.

$$E(P) = \frac{S(P)}{P} \quad (2.4)$$

$$= \frac{T_{total}(1)}{P\, T_{total}(P)} \quad (2.5)$$

Ideally, we would want the speedup to be equal to P, the number of PEs. This is sometimes called *perfect linear speedup*. Unfortunately, this is an ideal that can rarely be achieved because times for setup and finalization are not improved by adding more PEs, limiting the speedup. The terms that cannot be run concurrently are called the *serial terms*. Their running times represent some fraction of the total, called the *serial fraction*, denoted γ.

$$\gamma = \frac{T_{setup} + T_{finalization}}{T_{total}(1)} \quad (2.6)$$

The fraction of time spent in the parallelizable part of the program is then $(1 - \gamma)$. We can thus rewrite the expression for total computation time with P PEs as

$$T_{total}(P) = \gamma\, T_{total}(1) + \frac{(1-\gamma)\, T_{total}(1)}{P} \quad (2.7)$$

Now, rewriting S in terms of the new expression for $T_{total}(P)$, we obtain the famous Amdahl's law:

$$S(P) = \frac{T_{total}(1)}{(\gamma + \frac{1-\gamma}{P})\, T_{total}(1)} \quad (2.8)$$

$$= \frac{1}{\gamma + \frac{1-\gamma}{P}} \quad (2.9)$$

Thus, in an ideal parallel algorithm with no overhead in the parallel part, the speedup should follow Eq. 2.9. What happens to the speedup if we take our ideal parallel algorithm and use a very large number of processors? Taking the limit as

P goes to infinity in our expression for S yields

$$S = \frac{1}{\gamma} \tag{2.10}$$

Eq. 2.10 thus gives an upper bound on the speedup obtainable in an algorithm whose serial part represents γ of the total computation.

These concepts are vital to the parallel algorithm designer. In designing a parallel algorithm, it is important to understand the value of the serial fraction so that realistic expectations can be set for performance. It may not make sense to implement a complex, arbitrarily scalable parallel algorithm if 10% or more of the algorithm is serial—and 10% is fairly common.

Of course, Amdahl's law is based on assumptions that may or may not be true in practice. In real life, a number of factors may make the actual running time longer than this formula implies. For example, creating additional parallel tasks may increase overhead and the chances of contention for shared resources. On the other hand, if the original serial computation is limited by resources other than the availability of CPU cycles, the actual performance could be much better than Amdahl's law would predict. For example, a large parallel machine may allow bigger problems to be held in memory, thus reducing virtual memory paging, or multiple processors each with its own cache may allow much more of the problem to remain in the cache. Amdahl's law also rests on the assumption that for any given input, the parallel and serial implementations perform exactly the same number of computational steps. If the serial algorithm being used in the formula is not the best possible algorithm for the problem, then a clever parallel algorithm that structures the computation differently can reduce the total number of computational steps.

It has also been observed [Gus88] that the exercise underlying Amdahl's law, namely running exactly the same problem with varying numbers of processors, is artificial in some circumstances. If, say, the parallel application were a weather simulation, then when new processors were added, one would most likely increase the problem size by adding more details to the model while keeping the total execution time constant. If this is the case, then Amdahl's law, or fixed-size speedup, gives a pessimistic view of the benefits of additional processors.

To see this, we can reformulate the equation to give the speedup in terms of performance on a P-processor system. Earlier in Eq. 2.2, we obtained the execution time for T processors, $T_{total}(P)$, from the execution time of the serial terms and the execution time of the parallelizable part when executed on one processor. Here, we do the opposite and obtain $T_{total}(1)$ from the serial and parallel terms when executed on P processors.

$$T_{total}(1) = T_{setup} + PT_{compute}(P) + T_{finalization} \tag{2.11}$$

Now, we define the so-called *scaled serial fraction*, denoted γ_{scaled}, as

$$\gamma_{scaled} = \frac{T_{setup} + T_{finalization}}{T_{total}(P)} \tag{2.12}$$

and then

$$T_{total}(1) = \gamma_{scaled}T_{total}(P) + P(1 - \gamma_{scaled})T_{total}(P) \tag{2.13}$$

Rewriting the equation for speedup (Eq. 2.3) and simplifying, we obtain the scaled (or fixed-time) speedup.[1]

$$S(P) = P + (1 - P)\gamma_{scaled}. \tag{2.14}$$

This gives exactly the same speedup as Amdahl's law, but allows a different question to be asked when the number of processors is increased. Since γ_{scaled} depends on P, the result of taking the limit isn't immediately obvious, but would give the same result as the limit in Amdahl's law. However, suppose we take the limit in P *while holding* $T_{compute}$ *and thus* γ_{scaled} *constant.* The interpretation is that we are increasing the size of the problem so that the total running time remains constant when more processors are added. (This contains the implicit assumption that the execution time of the serial terms does not change as the problem size grows.) In this case, the speedup is linear in P. Thus, while adding more processors to solve a fixed problem may hit the speedup limits of Amdahl's law with a relatively small number of processors, if the problem grows as more processors are added, Amdahl's law will be pessimistic. These two models of speedup, along with a fixed-memory version of speedup, are discussed in [SN90].

2.6 COMMUNICATION

2.6.1 Latency and Bandwidth

A simple but useful model characterizes the total time for message transfer as the sum of a fixed cost plus a variable cost that depends on the length of the message.

$$T_{message-transfer} = \alpha + \frac{N}{\beta} \tag{2.15}$$

The fixed cost α is called *latency* and is essentially the time it takes to send an empty message over the communication medium, from the time the send routine is called to the time the data is received by the recipient. Latency (given in some appropriate time unit) includes overhead due to software and network hardware plus the time it takes for the message to traverse the communication medium. The *bandwidth* β (given in some measure of bytes per time unit) is a measure of the capacity of the communication medium. N is the length of the message.

The latency and bandwidth can vary significantly between systems depending on both the hardware used and the quality of the software implementing the communication protocols. Because these values can be measured with fairly simple benchmarks [DD97], it is sometimes worthwhile to measure values for α and β, as these can help guide optimizations to improve communication performance. For example, in a system in which α is relatively large, it might be worthwhile to try to

[1] This equation, sometimes known as Gustafson's law, was attributed in [Gus88] to E. Barsis.

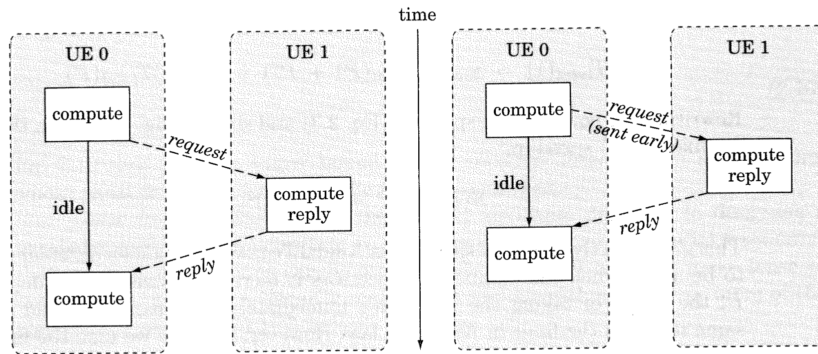

Figure 2.7: Communication without (left) and with (right) support for overlapping communication and computation. Although UE 0 in the computation on the right still has some idle time waiting for the reply from UE 1, the idle time is reduced and the computation requires less total time because of UE 1's earlier start.

restructure a program that sends many small messages to aggregate the communication into a few large messages instead. Data for several recent systems has been presented in [BBC+03].

2.6.2 Overlapping Communication and Computation and Latency Hiding

If we look more closely at the computation time within a single task on a single processor, it can roughly be decomposed into computation time, communication time, and idle time. The communication time is the time spent sending and receiving messages (and thus only applies to distributed-memory machines), whereas the idle time is time that no work is being done because the task is waiting for an event, such as the release of a resource held by another task.

A common situation in which a task may be idle is when it is waiting for a message to be transmitted through the system. This can occur when sending a message (as the UE waits for a reply before proceeding) or when receiving a message. Sometimes it is possible to eliminate this wait by restructuring the task to send the message and/or post the receive (that is, indicate that it wants to receive a message) and then continue the computation. This allows the programmer to overlap communication and computation. We show an example of this technique in Fig. 2.7. This style of message passing is more complicated for the programmer, because the programmer must take care to wait for the receive to complete after any work that can be overlapped with communication is completed.

Another technique used on many parallel computers is to assign multiple UEs to each PE, so that when one UE is waiting for communication, it will be possible to context-switch to another UE and keep the processor busy. This is an example of latency hiding. It is increasingly being used on modern high-performance computing systems, the most famous example being the MTA system from Cray Research [ACC+90].

2.7 SUMMARY

This chapter has given a brief overview of some of the concepts and vocabulary used in parallel computing. Additional terms are defined in the glossary. We also discussed the major programming environments in use for parallel computing: OpenMP, MPI, and Java. Throughout the book, we will use these three programming environments for our examples. More details about OpenMP, MPI, and Java and how to use them to write parallel programs are provided in the appendixes.