"I suspect that concurrency is best supported by a library and that such a library can be implemented without major language extensions."

—*Bjarne Stroustrup, inventor of C++*

In this Chapter

What is Concurrency? • The Benefits of Parallel Programming • The Benefits of Distributed Programming • The Minimal Effort Required • The Basic Layers of Software Concurrency • No Keyword Support for Parallelism in C++ • Programming Environments for Parallel and Distributed Programming • Summary—Toward Concurrency

The software development process now requires a working knowledge of parallel and distributed programming. The requirement for a piece of software to work properly over the Internet, on an intranet, or over some network is almost universal. Once the piece of software is deployed in one or more of these environments it is subjected to the most rigorous of performance demands. The user wants instantaneous and reliable results. In many situations the user wants the software to satisfy many requests at the same time. The capability to perform multiple simultaneous downloads of software and data from the Internet is a typical expectation of the user. Software designed to broadcast video must also be able to render graphics and digitally process sound seamlessly and without interruption. Web server software is often subjected to hundreds of thousands of hits per day. It is not uncommon for frequently used e-mail servers to be forced to survive the stress of a million sent and received messages during business hours. And it's not just the quantity of the messages that can require tremendous work, it's also the content. For instance, data transmissions containing digitized music, movies, or graphics devour network bandwidth and can inflict a serious penalty on server software that has not been properly designed. The typical

computing environment is networked and the computers involved have multiple processors. The more the software does, the more it is required to do. To meet the minimal user's requirements, today's software must work harder and smarter. Software must be designed to take advantage of computers that have multiple processors. Since networked computers are more the rule than the exception, software must be designed to correctly and effectively run, with some of its pieces executing simultaneously on different computers. In some cases, the different computers have totally different operating systems with different network protocols! To accommodate these realities, a software development repertoire must include techniques for implementing concurrency through parallel and distributed programming.

1.1 What is Concurrency?

Two events are said to be concurrent if they occur within the same time interval. Two or more tasks executing over the same time interval are said to execute concurrently. For our purposes, concurrent doesn't necessarily mean at the same exact instant. For example, two tasks may occur concurrently within the same second but with each task executing within different fractions of the second. The first task may execute for the first tenth of the second and pause, the second task may execute for the next tenth of the second and pause, the first task may start again executing in the third tenth of a second, and so on. Each task may alternate executing. However, the length of a second is so short that it appears that both tasks are executing simultaneously. We may extend this notion to longer time intervals. Two programs performing some task within the same hour continuously make progress of the task during that hour, although they may or may not be executing at the same exact instant. We say that the two programs are executing concurrently for that hour. Tasks that exist at the same time and perform in the same time period are concurrent. Concurrent tasks can execute in a single or multiprocessing environment. In a single processing environment, concurrent tasks exist at the same time and execute within the same time period by context switching. In a multiprocessor environment, if enough processors are free, concurrent tasks may execute at the same instant over the same time period. The determining factor for what makes an acceptable time period for concurrency is relative to the application.

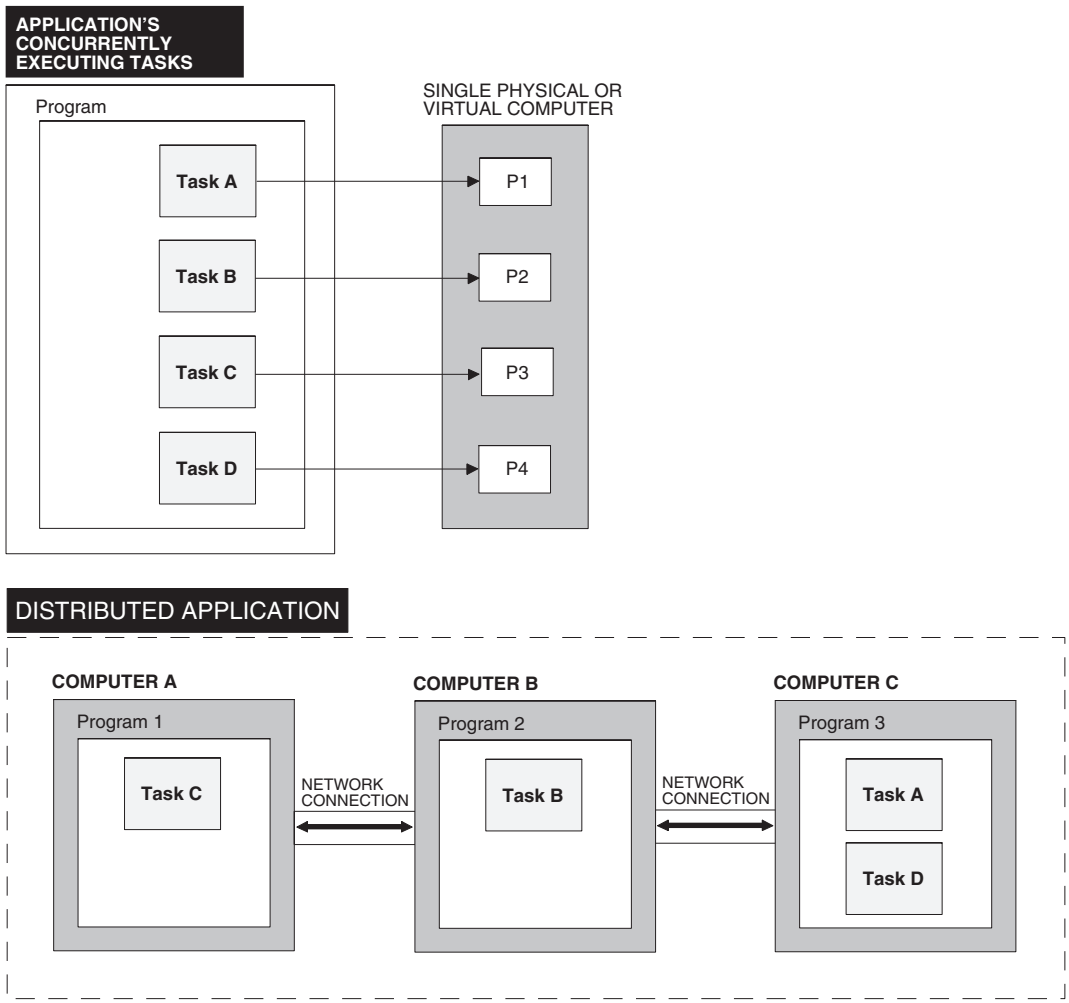
Concurrency techniques are used to allow a computer program to do more work over the same time period or time interval. Rather than designing the program to do one task at a time, the program is broken down in such a way that some of the tasks can be executed concurrently. In some situations, doing more work over the same time period is not the goal. Rather, simplifying the programming solution is the goal. Sometimes it makes more sense to think of the solution to the problem as a set of concurrently executed tasks. For instance, the solution to the problem of losing weight is best thought of as concurrently executed tasks: diet and exercise. That is, the improved diet and exercise regimen are supposed to occur over the same time interval (*not necessarily at the same instant*). It is typically not very beneficial to do one during one time period and the other within a totally different time period. The concurrency of both processes is the natural form of the solution. Sometimes concurrency is used to make software faster or get done with its work sooner. Sometimes concurrency is used to make software do more work over the same interval where speed is secondary to capacity. For instance, some web sites want customers to stay logged on as long as possible. So it's not how fast they can get the customers on and off of the site that is the concern—it's how many customers the site can support concurrently. So the goal of the software design is to handle as many connections as possible for as long a time period as possible. Finally, concurrency can be used to make the software simpler. Often, one long, complicated sequence of operations can be implemented easier as a series of small, concurrently executing operations. Whether concurrency is used to make the software faster, handle larger loads, or simplify the programming solution, the main object is software improvement using concurrency to make the software better.

1.1.1 The Two Basic Approaches to Achieving Concurrency

Parallel programming and distributed programming are two basic approaches for achieving concurrency with a piece of software. They are two different programming paradigms that sometimes intersect. *Parallel programming techniques* assign the work a program has to do to two or more processors within a single physical or a single virtual computer. *Distributed programming techniques* assign the work a program has to do to two or more processes—where the processes may or may not exist on the same computer.

That is, the parts of a distributed program often run on different computers connected by a network or at least in different processes. A program that contains parallelism executes on the same physical or virtual computer. The parallelism within a program may be divided into *processes* or *threads*. We discuss processes in Chapter 3 and threads in Chapter 4. For our purposes, distributed programs can only be divided into processes. Multithreading is restricted to parallelism. Technically, parallel programs are sometimes distributed, as is the case with PVM (Parallel Virtual Machine) programming. Distributed programming is sometimes used to implement parallelism, as is

Figure 1–1 Typical architecture for a parallel and distributed program.



the case with MPI (Message Passing Interface) programming. However, not all distributed programs involve parallelism. The parts of a distributed program may execute at different instances and over different time periods. For instance, a software calendar program might be divided into two parts: One part provides the user with a calendar and a method for recording important appointments and the other part provides the user with a set of alarms for each different type of appointment. The user schedules the appointments using part of the software, and the other part of the software executes separately at a different time. The alarms and the scheduling component together make a single application but they are divided into two separately executing parts. In pure parallelism, the concurrently executing parts are all components of the same program. In distributed programs, the parts are usually implemented as separate programs. Figure 1–1 shows the typical architecture for a parallel and distributed program.

The parallel application in Figure 1–1 consists of one program divided into four tasks. Each task executes on a separate processor, therefore, each task may execute simultaneously. The tasks can be implemented by either a process or a thread. On the other hand, the distributed application in Figure 1-1 consists of three separate programs with each program executing on a separate computer. Program 3 consists of two separate parts that execute on the same computer. Although Task A and D of Program 3 are on the same computer, they are distributed because they are implemented by two separate processes. Tasks within a parallel program are more tightly coupled than tasks within a distributed program. In general, processors associated with distributed programs are on different computers, whereas processors associated with programs that involve parallelism are on the same computer. Of course, there are hybrid programs that are both parallel and distributed. These hybrid combinations are becoming the norm.

1.2 The Benefits of Parallel Programming

Programs that are properly designed to take advantage of parallelism can execute faster than their sequential counterparts, which is a market advantage. In other cases the speed is used to save lives. In these cases *faster* equates to *better*. The solutions to certain problems are represented more naturally as a collection of simultaneously executing tasks. This is especially the case in many areas of scientific, mathematical, and artificial intelligence programming. This

means that parallel programming techniques can save the software developer work in some situations by allowing the developer to directly implement data structures, algorithms, and heuristics developed by researchers. Specialized hardware can be exploited. For instance, in high-end multimedia programs the logic can be distributed to specialized processors for increased performance, such as specialized graphics chips, digital sound processors, and specialized math processors. These processors can usually be accessed simultaneously. Computers with MPP (Massively Parallel Processors) have hundreds, sometimes thousands of processors and can be used to solve problems that simply cannot realistically be solved using sequential methods. With MPP computers, it's the combination of fast with pure *brute force* that makes the impossible possible. In this category would fall environmental modeling, space exploration, and several areas in biological research such as the Human Genome Project. Further parallel programming techniques open the door to certain software architectures that are specifically designed for parallel environments. For example, there are certain multiagent and blackboard architectures designed specifically for a parallel processor environment.

1.2.1 The Simplest Parallel Model (PRAM)

The easiest method for approaching the basic concepts in parallel programming is through the use of the PRAM (Parallel Random Access Machine). The PRAM is a simplified theoretical model where there are *n* processors labeled as P_1, P_2, P_3, . . . P_n and each processor shares one global memory. Figure 1–2 shows a simple PRAM.

All the processors have read and write access to a shared global memory. In the PRAM the access can be simultaneous. The assumption is that each processor can perform various arithmetic and logical operations in parallel. Also, each of the theoretical processors in Figure 1–2 can access the global shared memory in one *uninterruptible* unit of time. The PRAM model has

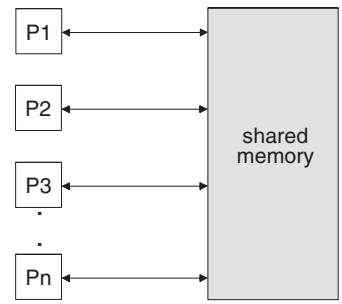


Figure 1–2
A Simple PRAM.

both concurrent and exclusive read algorithms. Concurrent read algorithms are allowed to read the same piece of memory simultaneously with no data corruption. Exclusive read algorithms are used to ensure that no two processors ever read the same memory location at the same time. The PRAM model also has both concurrent and exclusive write algorithms. Concurrent write algorithms allow multiple processors to write to memory, while exclusive write algorithms ensure that no two processors write to the same memory at the same time. Table 1–1 shows the four basic types of algorithms that can be derived from the read and write possibilities.

Table 1–1 Four Basic Read-Write Algorithms

Read-Write Algorithm Types	*Meaning*
EREW	Exclusive read exclusive write
CREW	Concurrent read exclusive write
ERCW	Exclusive read concurrent write
CRCW	Concurrent read concurrent write

We will refer to these algorithm types often in this book as we discuss methods for implementing concurrent architectures. The blackboard architecture is one of the important architectures that we implement using the PRAM model and it is discussed in Chapter 13. It is important to note that although PRAM is a simplified theoretical model, it is used to develop practical programs, and these programs can compete on performance with programs that were developed using more sophisticated models of parallelism.

1.2.2 The Simplest Parallel Classification

The PRAM gives us a simple model for thinking about how a computer can be divided into processors and memory and gives us some ideas for how those processors may access memory. A simplified scheme for classifying the parallel computers was introduced by M.J. Flynn.[1] These schemes were SIMD (Single Instruction Multiple Data) and MIMD (Multiple Instruction Multiple Data). These were later extended to SPMD (Single Program Multiple Data) and MPMD (Multiple Program Multiple Data). The SPMD (SIMD) scheme

[1]M.J. Flynn. Very high-speed computers. In Proceedings of the IEEE, 54, 1901–1909 (December 1966).

allows multiple processors to execute the same instruction or program with each processor accessing different data. The MPMD (MIMD) scheme allows for multiple processors with each executing different programs or instructions and each with its own data. So in one scheme all the processors execute the same program or instructions and in the other scheme each processor executes different instructions. Of course, there are hybrids of these models where the processors are divided up and some are SPMD and some are MPMD. Using SPMD, all of the processors are simply doing the same thing only with different data. For example, we can divide a single puzzle up into groups and assign each group to a separate processor. Each processor will apply the same rules for trying to put together the puzzle, but each processor has different pieces to work with. When all of the processors are done putting their pieces together, we can see the whole. Using MPMD, each processor executes something different. Even though they are all trying to solve the same problem, they have been assigned a different aspect of the problem. For example, we might divide the work of securing a Web server as a MPMD scheme. Each processor is assigned a different task. For instance, one processor monitors the ports, another processor monitors login attempts, another processor analyzes packet contents, and so on. Each processor works with its own data relative to its area of concern. Although the processors are each doing different work using different data, they are working toward a single solution: security. The parallel programming concepts that we discuss in this book are easily described using PRAM, SPMD (SIMD), and MPMD (MIMD). In fact, these schemes and models are used to implement practical small- to medium-scale applications and should be sufficient until you are ready to do advanced parallel programming.

1.3 The Benefits of Distributed Programming

Distributed programming techniques allow software to take advantage of resources located on the Internet, on corporate and organization intranets, and on networks. Distributed programming usually involves network programming in one form or another. That is, a program on one computer on a network needs some hardware or software resource that belongs to another computer either on the same network or on some remote network. Distributed programming is all about one program talking to another program over

some kind of network connection, which may involve everything from modems to satellites. The distinguishing feature of distributed programs is they are broken into parts. Those parts are usually implemented as separate programs. Those programs typically execute on separate computers and the program's parts communicate with each other over a network. Distributed programming techniques provide access to resources that may be geographically distant. For example, a distributed program divided into a Web server component and a Web client component can execute on two separate computers. The Web server component can be located in Africa and the Web client component can be located in Japan. The Web client part is able to use software and hardware resources of the Web server component, although they are separated by a great distance and almost certainly located on different networks running different operating environments. Distributed programming techniques provide shared access to expensive hardware and software resources. For instance, an expensive, high-end holographic printer may have print server software that provides print services to client software. The print client software resides on one computer and the print server software resides on another computer. Only one print server is needed to serve many print clients. Distributed computing can be used for redundancy and fail over. If we divide the program up into a number of parts with each running on different computers, then we may assign some of the parts the same task. If one of the computers fails for some reason then another part of the same program executing on a different computer picks up the work. Databases can be used to hold billions, trillions, even quadrillions of pieces of information. It is simply not practical for every user to have a copy of the database. The problem is some users are located in different buildings than where the computer with the database is located. Some users are located in different cities, states, and in some instances, countries. Distributed programming techniques are used to allow users to share the massive database regardless of where they are located.

1.3.1 The Simplest Distributed Programming Models

The client-server model of distributed computing is perhaps the easiest to understand and the most commonly used. In this model, a program is divided up into two parts: One part is called the server and the other the client. The server has direct access to some hardware or software resource that the

client wants to use. In most cases, the server is located on a different machine than the client. Typically, there is a many-to-one relationship between the server and the client, that is, there is usually one server fulfilling the requests of many clients. The server usually mediates access to a large database, an expensive hardware resource, or an important collection of applications. The client makes requests for data, calculations, and other types of processing. A search engine is a good example of a client-server application. Search engines are used to locate information on the Internet or on corporate and organization intranets. The client is used to obtain a keyword or phrase that the user is interested in. The client software part then passes the request to the server software part. The server has the muscle to perform the massive search for the user's keyword or phrase. The server has either direct access to the information or to other servers that have access to the information. Ideally, the server finds the keyword or phrase the user requested and returns that information to the client component. Although the client and the server are separate programs on separate computers, they make up a single application. This division of a piece of software into a client and a server is the primary method of distributed programming. The client-server model also has other forms depending on the environment. For instance, the term *producer-consumer* is a close cousin of client-server. Typically, client-server applications refer to larger programs and producer-consumer refers to smaller programs. Usually when the programs are at the operating system level or lower they are called producer-consumer, and when they are above the operating system level they are usually called client-server (however, there are always exceptions).

1.3.2 The Multiagent (Peer-to-Peer) Distributed Model

Although the client-server model is the most prevalent distributed programming model in use, it is not the only model. Agents are rational software components that are self directed, often autonomous, and can continuously execute. Agents can both make requests of other software components and fulfill requests of other software components. Agents can cooperate within groups to perform certain tasks collectively. In this model there is no specific client or server. The agents form a kind of peer-to-peer model where each of the components are on somewhat equal footing and each component has something to offer to the other. For example, an agent that is providing a

price quote for the refurbishing of a vintage sports car might work together with other agents. Where one agent specializes in engine work, another specializes in body work, another specializes in interior design and so on. These agents may cooperatively and collectively come up with the most competitive quote for refurbishing the car. The agents are distributed because each agent is located on a different server on the Internet. The agents use an agreed-upon Internet protocol to communicate. The client-server model is a natural fit for certain types of distributed programming and the peer-to-peer agent model is a natural fit for certain types of distributed programming. We explore both types in this book. The client-server and peer-to-peer models can be used to satisfy most distributed programming demands.

1.4 The Minimal Effort Required

Parallel and distributed programming come with a cost. Although there are many benefits to writing parallel and distributed programming, there are also some challenges and prerequisites. We discuss some challenges in Chapter 2. We mention the prerequisites here. Before a program is written or a piece of software is developed, it must first go through a design process. For parallel and distributed programs, the design process will include three issues: decomposition, communication, and synchronization.

1.4.1 Decomposition

Decomposition is the process of dividing up the problem and the solution into parts. Sometimes the parts are grouped into logical areas (i.e., searching, sorting, calculating, input, output, etc.). In other situations the parts are grouped by logical resource (i.e., file, communication, printer, database, etc.). The decomposition of the software solution amounts to the WBS (work breakdown structure). The WBS determines which piece of software does what. One of the primary issues of concurrent programming is identifying a natural WBS for the software solution at hand. There is no simple or cookbook approach to identifying the WBS. Software development is the process of translating concepts, ideas, patterns of work, rules, algorithms, or formulas into sets of instructions and data that can be executed or manipulated by a computer. This is largely a process of modeling. Software models are repro-

ductions in software of some real-world task, process, or ideal. The purpose of the model is to imitate or duplicate the behavior and characteristics of some real-world entity in a particular domain. This process of modeling uncovers the natural WBS of a software solution. The better the model is understood and developed the more natural the WBS will be. Our approach is to uncover the parallelism or distribution within a solution through modeling. If parallelism doesn't naturally fit, don't force it. The question of how to break up an application into concurrently executing parts should be answered during the design phase and should be obvious in the model of the solution. If the model of the problem and the solution don't imply or suggest parallelism and distribution then try a sequential solution. If the sequential solution fails, that failure may give clues to how to approach the parallelism.

1.4.2 Communication

Once the software solution is decomposed into a number of concurrently executing parts, those parts will usually do some amount of communicating. How will this communication be performed if the parts are in different processes or different computers? Do the different parts need to share any memory? How will one part of the software know when the other part is done? Which part starts first? How will one component know if another component has failed? These issues have to be considered when designing parallel or distributed systems. If no communication is required between the parts, then the parts don't really constitute a single application.

1.4.3 Synchronization

The WBS designates who does what. When multiple components of software are working on the same problem, they must be coordinated. Some component has to determine when a solution has been reached. The components' order of execution must be coordinated. Do all of the parts start at the same time or does some work while others wait? What two or more components need access to the same resource? Who gets it first? If some of the parts finish their work long before the other parts, should they be assigned new work? Who assigns the new work in such cases? DCS (decomposition, communication, and synchronization) is the minimum that must be considered when approaching parallel or distributed programming. In addition to considering DCS, the location of DCS is also important. There are several layers

of concurrency in application development. DCS is applied a little differently in each layer.

1.5 The Basic Layers of Software Concurrency

In this book we are concerned with concurrency within the application as opposed to concurrency at the operating system level, or concurrency within hardware. Although the concurrency within hardware and the concurrency at the operating system level support application concurrency, our focus is on the application. For our purposes, concurrency occurs at:

- Instruction level
- Routine (function/procedure) level
- Object level
- Application level

1.5.1 Concurrency at the Instruction Level

Concurrency at the instruction level occurs when multiple parts of a single instruction can be executed simultaneously. Figure 1–3 shows how a single instruction can be decomposed for simultaneous execution.

In Figure 1-3, the component (A + B) can be executed at the same time as (C – D). This is an example of concurrency at the instruction level. This kind of parallelism is normally supported by compiler directives and is not under the direct control of a C++ programmer.

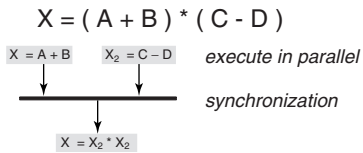


Figure 1–3
Decomposition of a single instruction.

1.5.2 Concurrency at the Routine Level

The WBS structure of a program may be along function lines, that is, the total work involved in a software solution is divided between a number of functions. If these functions are assigned to threads, then each function can execute on a different processor and if enough processors are available, each function can execute simultaneously. We discuss threads in more detail in Chapter 4.

1.5.3 Concurrency at the Object Level

The WBS of a software solution may be distributed between objects. Each object can be assigned to a different thread, or process. Using the CORBA (Common Object Request Broker Architecture) standard, each object may be assigned to a different computer on the network or different computer on a different network. We discuss CORBA in more detail in Chapter 8. Objects residing in different threads or processes may execute their methods concurrently.

1.5.4 Concurrency of Applications

Two or more applications can cooperatively work together to solve some problem. Although the application may have originally been designed separately and for different purposes, the principles of code reuse often allow applications to cooperate. In these circumstances two separate applications work together as a single distributed application. For example, the Clipboard was not designed to work with any one application but can be used by a variety of applications on the desktop. Some uses of the Clipboard had not been dreamed of during its original design.

The second and the third layers are the primary layers of concurrency that we will focus on in this book. We show techniques for implementing concurrency in these layers. Operating system and hardware issues are presented only where they are necessary in the context of application design. Once we have an appropriate WBS for a parallel programming or distributed programming design, the question is how do we implement it in C++.

1.6 No Keyword Support for Parallelism in C++

The C++ language does not include any keyword primitives for parallelism. The C++ ISO standard is for the most part silent on the topic of multithreading. There is no way within the language to specify that two or more statements should be executed in parallel. Other languages use built-in parallelism as a selling feature. Bjarne Stroustrup, the inventor of the C++ language, had something else in mind. In Stroustrup's opinion:

> It is possible to design concurrency support libraries that approach built-in concurrency support both in convenience and efficiency. By relying on libraries, you can support a variety of concurrency models, though, and thus serve the users that need those different models better than can be done by a single built-in concurrency model. I expect this will be the direction taken by most people and that the portability problems that arise when several concurrency-support libraries are used within the community can be dealt with by a thin layer of interface classes.

Furthermore, Stroustrup says, "I recommend parallelism be represented by libraries within C++ rather than as a general language feature." The authors have found Stroustrup's position and recommendation on parallelism as a library the most practical option. This book is only made possible because of the availability of high-quality libraries that can be used for parallel and distributed programming. The libraries that we use to enhance C++ implement national and international standards for parallelism and distributed programming and are used by thousands of C++ programmers worldwide.

1.6.1 The Options for Implementing Parallelism Using C++

Although there are special versions of C++ that implement parallelism, we present methods on how parallelism can be implemented using the ISO (International Standard Organization) standard for C++. The library approach to parallelism is the most flexible. System libraries and user-level libraries can be used to support parallelism in C++. System libraries are those libraries provided by the operating system environment. For example, the POSIX threads library is a set of system calls that can be used in conjunction

with C++ to support parallelism. The POSIX (Portable Operating System Interface) threads are part of the new Single UNIX Specification. The POSIX threads are included in the IEEE Std. 1003.1-2001. The Single UNIX Specification is sponsored by the Open Group and developed by the Austin Common Standards Revision Group. According to the Open Group, the Single UNIX Specification is:

- Designed to give software developers a single set of APIs to be supported by every UNIX system.
- Shifts the focus from incompatible UNIX system product implementations to compliance to a single set of APIs.
- It is the codification and dejure standardization of the common core of UNIX system practice.
- The basic objective is portability of both programmers and application source code.

The Single UNIX Specification Version 3 includes the IEEE Std 1003.1-2001 and the Open Group Base Specifications Issue 6. The IEEE POSIX standards are now a formal part of the Single UNIX Specification and vice versa. There is now a single international standard for a portable operating system interface. C++ developers benefit because this standard contains APIs for creating threads and processes. Excluding instruction-level parallelism, dividing a program up into either threads or processes is the only way to achieve parallelism with C++. The new standard provides the tools to do this. The developer can use:

- POSIX threads (also referred to as pthreads)
- POSIX spawn function
- the `exec()` family of functions

These are all supported by system API calls and system libraries. If an operation system complies with the Single UNIX Specification Version 3, then these APIs will be available to the C++ developer. These APIs are discussed in Chapters 3 and 4. They are used in many of the examples in this book. In addition to system-level libraries, user-level libraries that implement other international standards such as the MPI (Message Passing Interface), PVM (Parallel Virtual Machine), and CORBA (Common Object Request Broker Architecture) can be used to support parallelism with C++.

1.6.2 MPI Standard

The MPI is the standard specification for message passing. The MPI was designed for high performance on both massively parallel machines and on workstation clusters. This book uses the MPICH implementation of the MPI standard. MPICH is a freely available, portable implementation of MPI. The MPICH provides the C++ programmer with a set of APIs and libraries that support parallel programming. The MPI is especially useful for SPMD and MPMD programming. The authors use the MPICH implementation of MPI on a 32-node cluster running Linux and an 8-node cluster running Solaris and Linux. Although C++ doesn't have parallel primitives built in, it can take advantage of power libraries such as MPICH that does support parallelism. This is one of the benefits of C++. It is designed for flexibility.

1.6.3 PVM: A Standard for Cluster Programming

The PVM is a software package that permits a heterogeneous collection of computers hooked together by a network to be used as a single large parallel computer. The overall objective of the PVM system is to enable a collection of computers to be used cooperatively for concurrent or parallel computation. A PVM library implementation supports:

- Heterogeneity in terms of machines, networks, and applications
- Explicit message-passing model
- Process-based computation
- Multiprocessor support (MPP, SMP)
- Translucent access to hardware (applications can either ignore or take advantage of hardware differences)
- Dynamically configurable host pool (processors can be added and deleted at runtime and can include processor mixes)

The PVM is the easiest to use and most flexible environment available for basic parallel programming tasks that require the involvement of different types of computers running different operating systems. The PVM library is especially useful for several single processor systems that can be networked together to form a virtual parallel processor machine. We discuss techniques

for using PVM with C++ in Chapter 6. The PVM is the de facto standard for implementing heterogeneous clusters and is freely available and widely used. The PVM has excellent support for MPMD (MIMD) and SPMD (SIMD) models of parallel programming. The authors use PVM for small- to medium-size parallel programming tasks and the MPI for larger, more complex MPI tasks. PVM and MPI are both libraries that can be used with C++ to do cluster programming.

1.6.4 The CORBA Standard

CORBA is the standard for distributed cross-platform object-oriented programming. We mention CORBA here under parallelism because implementations of the CORBA standard can be used to develop multiagent systems. Multiagent systems offer important models of peer-to-peer distributed programming. Multiagent systems can work concurrently. This is one of the areas where parallel programming and distributed programming overlap. Although the agents are executing on different computers, they are executing during the same time period, working cooperatively on a common problem. The CORBA standard provides an open, vendor-independent architecture and infrastructure that computer applications use to work together over networks. Using the standard protocol IIOP, a CORBA-based program from any vendor, on almost any computer, operating system, programming language, and network, can interoperate with a CORBA-based program from the same or another vendor on almost any other computer operating system, programming language, and network. In this book we use the MICO implementation. MICO is a freely available and fully compliant implementation of the CORBA standard. MICO supports C++.

1.6.5 Library Implementations Based on Standards

MPICH, PVM, MICO, and POSIX threads are each library implementations based on standards. This means that software developers can rely on these implementations to be widely available and portable across multiple platforms. These libraries are freely available and used by software developers worldwide. The POSIX threads library can be used with C++ to do multithreaded programming. If the program is running on a computer that has multiple processors, then each thread can possibly run on a separate proces-

sor and thereby execute concurrently. If only a single processor is available, then the illusion of parallelism is provided and concurrency is achieved through the process of context switching. POSIX threads are perhaps the easiest way to introduce parallelism within a C++ program. Whereas the MPICH, PVM, and MICO libraries will have to be downloaded or obtained (they are readily available), any operating system environment that is client with the POSIX standard or the new UNIX Specification Version 3 will have a POSIX threads implementation. Each library offers a slightly different model of parallelism. Table 1–2 shows how each library can be used with C++.

Table 1–2 MPICH, PVM, MICO, and POSIX Threads Used with C++

Libraries	*C++ Usage*
MPICH	Supports large-scale, complex cluster programming. Strong support for SPMD model. Also supports SMP, MPP, and multiuser configurations.
PVM	Supports cluster programming of heterogeneous environments. Easy to use for single-user, small to medium cluster applications. Also supports MPP.
MICO	Supports either distributed or object-oriented parallel programming. Contains nice support for agent and multiagent programming.
POSIX	Supports parallel processing within a single application at the function or object level. Can be used to take advantage of SMP or MPP.

Whereas languages that depend on built-in support for parallelism are restricted to the models supplied, the C++ developer is free to mix and match parallel programming models. As the nature of the applications change, a C++ developer can select different libraries to match the scenario.

1.7 Programming Environments for Parallel and Distributed Programming

The most common environments for parallel and distributed programming are clusters, MPPs, and SMP computers.

Clusters are collections of two or more computers that are networked together to provide a single, logical system. The group of computers appear to

the application as a single virtual computer. MPP (Massively Parallel Processors) is a single computer that has hundreds of processors. SMP (Symmetric Multiprocessing) is a single system that has processors that are tightly coupled where the processors share memory and the data path. SMP processors share the resources and are all controlled by a single operating system. This book provides a gentle introduction to parallel and distributed programming, therefore we focus our attention on small clusters of 8 to 32 processors and on multiprocessor machines with 2 to 4 processors. Although many of the techniques we discuss can be used in MPP environments or in large SMP environments, our primary attention is on moderate systems.

Summary—Toward Concurrency

Throughout this book we present an architectural approach to parallel and distributed programming. The emphasis is placed on uncovering the natural parallelism within a problem and its solution. This parallelism is captured within the software model for the solution. We suggest object-oriented methods to help manage the complexity of parallel and distributed programming. Our mantra is *function follows form*. We use the library approach to provide parallelism support for the C++ language. The libraries we recommend are based on national and international standards. Each library is freely available and widely used. Techniques and concepts presented in the book are vendor independent, nonproprietary, and rely on open standards and open architectures. The C++ programmer and software developer can use different parallel models to serve different needs because each parallelism model is captured within a library. The library approach to parallel and distributed programming gives the C++ programmer the greatest possible flexibility. While parallel and distributed programming can be fun and rewarding, it presents several challenges. In the next chapter we will provide an overview of the most common challenges to parallel and distributed programming.