The principal goal of this chapter is to introduce the common issues that a programmer faces when implementing a parallel application. The treatment assumes that the reader is familiar with programming a uniprocessor using a conventional language, such as Fortran. The principal challenge of parallel programming is to decompose the program into subcomponents that can be run in parallel. However, to understand some of the low-level issues of decomposition, the programmer must have a simplified view of parallel machine architecture. Thus, we begin our treatment with a review of this topic, with the goal of identifying the characteristics that are most important for the parallel programmer to understand. This discussion, found in Section 3.1, focuses on two main parallel machine organizations—shared memory and distributed memory—that characterize most current machines. The section also treats hybrids of the two main memory designs.

The standard parallel architectures support a variety of decomposition strategies, such as decomposition by task (task parallelism) and decomposition by data (data parallelism). Our introductory treatment will concentrate on data parallelism because it represents the most common strategy for scientific programs on parallel machines. In data parallelism, the application is decomposed by subdividing the data space over which it operates and assigning different processors to the work associated with different data subspaces. Typically this strategy involves some data sharing at the boundaries, and the programmer is responsible for ensuring that this data sharing is handled correctly—that is, data computed by one processor and used by another are correctly synchronized.

Once a specific decomposition strategy is chosen, it must be implemented. Here, the programmer must choose the programming model to use. The two most common models are the following:
• The *shared-memory model*, in which it is assumed that all data structures are allocated in a common space that is accessible from every processor.

• The *message-passing model*, in which each processor (or process) is assumed to have its own private data space, and data must be explicitly moved between spaces as needed.

In the message-passing model, data structures are distributed across the processor memories; if a processor needs to use a data item that is not stored locally, the processor that owns that data item must explicitly "send" it to the requesting processor. The latter must execute an explicit "receive" operation, which is synchronized with the send, before it can use the communicated data item. These issues are discussed in Section 3.2.

To achieve high performance on parallel machines, the programmer must be concerned with *scalability* and *load balance*. Generally, an application is thought to be scalable if larger parallel configurations can solve proportionally larger problems in the same running time as smaller problems on smaller configurations. To understand this issue, we introduce in Section 3.3.1 a formula that defines parallel *speedup* and explore its implications. Load balance typically means that the processors have roughly the same amount of work, so that no one processor holds up the entire solution. To balance the computational load on a machine with processors of equal power, the programmer must divide the work and communications evenly. This can be challenging in applications applied to problems that are unknown in size until run time.

A particular bottleneck on most parallel machines is the performance of the memory hierarchy, both on a single node and across the entire machine. In Section 3.4, we discuss various strategies for enhancing the reuse of data by a single processor. These strategies typically involve some sort of loop "blocking" or "strip mining," so that whole subcomputations fit into cache.

Irregular or adaptive problems present special challenges for parallel machines because it is difficult to maintain load balance when the size of subproblems is unknown until run time or if the problem size may change after execution begins. Special methods involving run-time reconfiguration of a computation are required to deal with these problems. These methods are discussed in Section 3.3.3.

Several aspects of programming parallel machines are much more complicated than their counterparts for sequential systems. Parallel debugging, for example, must deal with the possibilities of race conditions or out-of-order execution (see Section 3.5). Performance analysis and tuning must deal with the especially challenging problems of detecting load imbalances and communication bottlenecks (see Section 3.6). In addition, it must present diagnostic information to the user in a format that is related to the program structure and programming model. Finally, input/output on parallel machines, particularly those with distributed memory, presents problems of how to read files that are distributed across disks in a system into memories that are distributed with the processors (see Section 3.7).
Figure 3.1 A uniform-access shared-memory architecture.

These topics do not represent all the issues of parallel programming. We hope, however, that a discussion of them will convey some of the terminology and intuition of parallel programming. In so doing, it will set the stage for the remainder of this book.

3.1 Architectural Considerations

Chapter 2 provided a detailed review of parallel computer architectures. In this chapter, we provide a simple introduction to these topics that covers most of the important issues needed to understand parallel programming.

First, as discussed in Chapter 2, we observe that most of the modern parallel machines fall into two basic categories:

1. *Shared-memory machines*, which have a single shared address space that can be accessed by any processor.
2. *Distributed-memory machines*, in which the system memory is packaged with individual nodes of one or more processors and communication is required to provide data from the memory of one processor to a different processor.

3.1.1 Shared Memory

The organization of a shared-memory machine is depicted in Figure 2.5. Figure 3.1 shows a slightly more detailed diagram of a shared-memory system with four processors, each with a private cache, interconnected to a global shared memory via a single system bus. This organization is typically called a *symmetric multiprocessor* (SMP).

In a symmetric multiprocessor, each processor can access all locations in global memory using standard load operations. The hardware ensures that the caches
are "coherent" by watching the system bus and invalidating cached copies of any block that is written into. This mechanism is generally invisible to the user, except when different processors are simultaneously attempting to write into the same cache line, which can cause the cache line to ping-pong between two different caches, a situation known as thrashing. To avoid this problem, the programmer and programming system must be careful with shared data structures and nonshared data structures that can be located on the same cache block, a situation known as false sharing. Synchronization of accesses to shared data structures is a major issue on shared-memory systems—it is up to the programmer to ensure that operations by different processors on a shared data structure leave that data structure in a consistent state. Various memory consistency models are discussed in Section 2.2.1.

The main problem with the shared-memory system as described above is that it is not scalable to large numbers of processors. Most bus-based systems are limited to 32 or fewer processors because of contention on the bus. If the bus is replaced by a crossbar switch, systems can scale to as many as 128 processors, although the cost of the switch increases as the square of the number of processors, making this organization impractical for truly large numbers of processors. Multistage switches can be made to scale better at the cost of longer latencies to memory.

3.1.2 Distributed Memory

The scalability limitations of shared memory have led designers to use distributed-memory organizations such as the one depicted in Figure 3.2. Here the global shared memory has been replaced by a smaller local memory attached to each processor. Communication among the processor-memory configurations is over an interconnection network. These systems can be made scalable if a scalable interconnection network is used. For example, a hypercube has cost proportional to \( n \lg(n) \) where \( n \) is the number of processors.

The advantage of a distributed-memory design is that access to local data can be quite fast. On the other hand, access to remote memories requires much more
effort. Most distributed-memory systems support a message-passing programming model, in which the processor owning a datum must send it to any processor that needs it. These “send–receive” communication steps typically incur long start-up times, although the bandwidth after start-up can be high. Hence, on message-passing systems, it typically pays to send fewer, longer messages.

The principal programming problem for distributed-memory systems is management of communication between processors. Usually this means consolidation of messages between the same pair of processors and overlapping communication and computation so that long latencies are hidden. In addition, data placement is important so that as few data references as possible require communication.

### 3.1.3 Hybrid Systems

As seen in Chapter 2, there are various ways in which the two memory paradigms are combined. Some distributed-memory machines allow a processor to directly access a datum in a remote memory. On these distributed shared-memory (DSM) systems, the latency associated with a load varies with the distance to the remote memory. Cache coherency on DSM systems is a complex problem that is usually handled by a sophisticated network interface unit. Given that DSM systems have longer access times to remote memory, data placement is an important programming consideration.

For very large parallel systems, a hybrid architecture called an SMP cluster is common. An SMP cluster looks like a distributed-memory system in which each of the individual components is a symmetric multiprocessor rather than a single processor node. This design permits high parallel efficiency within a multiprocessor node, while permitting systems to scale to hundreds or even thousands of processors. Programming for SMP clusters provides all the challenges of both shared- and distributed-memory systems. In addition, it requires careful thought about how to partition the parallelism within and between computational nodes.

### 3.1.4 Memory Hierarchy

As discussed in Chapter 2, the design of memory hierarchies is an integral part of the design of parallel computer systems because the memory hierarchy is a determining factor in the performance of the individual nodes in the processor array. A typical memory hierarchy is depicted in Figure 3.3. Here the processor and a level-1 (L1) cache memory are found on-chip, and a larger level-2 (L2) cache lies between the chip and the memory.

When a processor executes a load instruction, the L1 cache is first interrogated to determine if the desired datum is available. If it is, the datum can be delivered to the processor in two to five processor cycles. If the datum is not found in the L1 cache, the processor stalls while the L2 cache is interrogated. If the desired datum is found in L2, then the stall may last for only 10 to 20 cycles. If the datum is not found in either cache, a full cache miss is taken with a delay of possibly 100 cycles or
more. Whenever a miss occurs, the datum is saved in every cache in the hierarchy, if it is not already there. Note that on modern machines, caches transfer data in a minimum-size cache block, so that whenever a datum is loaded to that cache, the entire block containing that datum comes with it.

The performance of the memory hierarchy is determined by two hardware parameters: latency, which is the time required to fetch a desired datum from memory, and bandwidth, which is the number of bytes per unit time that can be delivered from the memory at full speed. Long latencies increase the cost of cache misses, thus slowing performance, while limited bandwidth can cause applications to become “memory bound,” that is, continuously stalled waiting for data. These two factors are complicated by the multilevel nature of memory hierarchies, because each level will have a different bandwidth and latency to the next level. For example, the SGI Origin 2000 can deliver about 4 bytes per machine cycle from the L1 cache to the processor and 4 bytes per cycle from the L2 cache to the L1 cache, but it can deliver only about 0.8 bytes per cycle from memory to L1 cache [272].

Another important parameter that affects memory performance on a uniprocessor is the length of the standard cache block (or cache line). Most cache systems will only transfer blocks of data between levels of the memory hierarchy. If all the data transferred in a block are used, then no bandwidth is wasted. In that case, the cost of the cache miss can be amortized over all the data in the block. If only one or two data items are used, then the average latency is much higher and the effective bandwidth much lower.

There are two kinds of strategies for overcoming latency problems. Latency hiding attempts to overlap the latency of a miss with computation. Prefetching of cache
lines is a latency-hiding strategy. *Latency tolerance*, on the other hand, attempts to restructure a computation to make it less subject to performance problems due to long latencies. The single most important latency tolerance technique is *cache blocking*, which brings accesses to the same locations closer together in time so that accesses after the first are likely to find the desired data in cache.

Strategies that improve reuse in cache also improve effective bandwidth utilization. Perhaps the most important way to ensure good bandwidth utilization is to organize data and computations to use all the items in a cache line whenever it is fetched from memory. Ensuring that computations access data arrays in strides of one is an example of how this might be done.

The memory hierarchies on parallel machines are more complicated because of the existence of multiple caches on shared-memory systems and the long latencies to remote memories on distributed-memory configurations. There may also be interference between data transfers between memories and from local memory to a processor.

### 3.2 Decomposing Programs for Parallelism

Given that you have decided to implement a program for a parallel machine, there are four main issues that you must deal with. First, you must have a way of identifying components of the computation that can safely be run in parallel. Second, you need to adopt a strategy for decomposing the program into parallel components. Third, you must actually write the parallel program, which requires that you choose a programming model and interface for the implementation. Finally, you must choose an implementation style that is effective for the given application and that works well with the chosen programming model. In this section, we discuss each of these issues and illustrate them with an extended example at the end.

#### 3.2.1 Identification of Parallelism

The first task in a parallel implementation is to identify the portions of the code where there is parallelism to exploit. To do this we must address a fundamental question: *When can we run two different computations in parallel?* We cannot answer this question without thinking about what it means for two computations to run in parallel. Most programmers think of the meaning of a program to be defined by the sequential implementation. That is, for a parallel implementation to be correct, it must produce the same answers as the sequential version every time it is run. So the question becomes: *When can we run two computations from a given sequential program in parallel and expect that the answers will be the same as those produced by the sequential program?* By “running in parallel,” we mean asynchronously, with synchronization at the end. Thus, the parallel version of the program will spawn a number of parallel processes to handle different computations, with each of the computations running until the end, when they synchronize.