1.16 Consider a machine running at 100 MIPS on some workload with the following mix: 50% ALU, 20% loads, 10% stores, and 20% branches. Suppose the instruction miss rate is 1%, the data miss rate is 5%, and the cache line size is 32 bytes. For the purpose of this calculation, treat a store miss as requiring two cache line transfers, one to load the newly updated line and one to replace the dirty line. If the machine provides a 250-MB/s bus, how many processors can it accommodate at peak bus bandwidth? What is the bandwidth demand of each processor?

1.17 Exercise 1.16 looks only at the sum of the average bandwidths. As the bus approaches saturation, however, it takes longer to obtain access for the bus, so it looks to the processor as if the memory system is slower. The effect is to slow down all of the processors in the system, thereby reducing their bandwidth demand. Let's try an analogous calculation from the other direction.

Assume the instruction mix and miss rate as in Exercise 1.16, but ignore the MIPS since that depends on the performance of the memory system. Assume instead that the processor runs at 100 MHz and has an ideal CPI (with a perfect memory system) of one. The unloaded cache miss penalty is 20 cycles. You can ignore the write back for stores. (As a starter, you might want to compute the MIPS rate for this new machine.) Assume that the memory system (i.e., the bus and the memory controller) is utilized throughout the miss. What is the utilization of the memory system U_1 with a single processor? From this result, estimate the number of processors that could be supported before the processor demand would exceed the available bus bandwidth.

1.18 Of course, no matter how many processors you place on the bus, they will never exceed the available bandwidth. Explain what happens to processor performance in response to bus contention. Can you formalize your observations?

2

Parallel Programs

To understand and evaluate design decisions in a parallel machine, we must have an idea of the software that runs on the machine. Understanding program behavior has led to some of the most important advances in uniprocessors, including memory hierarchies and instruction set design. It is all the more important in multiprocessors, both because of the increase in degrees of freedom and because of the much greater performance penalties caused by mismatches between applications and systems.

Understanding parallel software is important for algorithm designers, for programmers, and for architects. As algorithm designers, it helps us focus on designing algorithms that can be run effectively in parallel on real systems. As programmers, it helps us understand the key performance issues and obtain the best performance from a system. And as architects, it helps us understand the workloads we are designing against and their important degrees of freedom. Parallel software and its implications will be the focus of the next three chapters of this book. This chapter describes the process of creating parallel programs in the major programming models. Chapter 3 focuses on the performance issues that must be addressed in this process, exploring some of the key interactions between parallel applications and architectures. Chapter 4 relies on this understanding of hardware/software interactions to develop guidelines for using parallel workloads to evaluate architectural trade-offs. In addition to being helpful to architects, the material in these chapters is useful for users of parallel machines as well: Chapters 2 and 3 for programmers and algorithm designers, and Chapter 4 for users making decisions about what types of machines to procure. However, the major focus is on issues that architects should understand before they get into the nuts and bolts of machine design.

As architects of sequential machines, we generally take programs for granted: the field is mature, and there is a large base of programs that can (or must) be viewed as fixed. We optimize the machine design against the requirements of these programs. Although we recognize that programmers may further optimize their code—for example, as caches become larger or floating-point support is improved—we usually evaluate new designs without anticipating such software changes. Compilers may evolve along with the architecture, but the source program is still treated as fixed. In parallel architecture, there is a much stronger and more dynamic interaction between the evolution of machine designs and that of parallel software. Since parallel computing is all about performance, programming tends to be oriented toward taking advantage of what machines provide. Parallelism offers a new degree of

freedom—the number of processors—and higher costs for data access and coordination, giving the programmer a wide scope for software optimizations. Even as architects, we therefore need to open the application "black box." Understanding the important aspects of the process of creating parallel software (the focus of this chapter) helps us appreciate the role and limitations of the architecture. A deeper look at performance issues in the next chapter will shed greater light on hardware/software trade-offs.

Even after a problem and a good sequential algorithm to solve it are determined, a substantial process is involved in arriving at a parallel program and the execution characteristics that it offers to a multiprocessor architecture. This chapter presents general principles of the parallelization process and illustrates them with real examples. It begins by introducing four actual problems that serve as case studies throughout the next two chapters. Then it describes the four major steps in creating a parallel program—using the case studies to illustrate—followed by examples of how a simple parallel program might be written in each of the major programming models. As discussed in Chapter 1, the dominant models from a programming perspective narrow down to three: the data parallel model, a shared address space or shared memory, and message passing between private address spaces. This chapter illustrates the primitives provided by these models and how they might be used, without much concern for performance. After the performance issues in the parallelization process are understood in Chapter 3, the four application case studies will be treated in more detail to create high-performance versions of them.

2.1 PARALLEL APPLICATION CASE STUDIES

We saw in the previous chapter that multiprocessors are used for a wide range of applications—from multiprogramming and commercial computing to so-called Grand Challenge scientific problems—and that the most demanding of these applications tend to be from scientific and engineering computing. Of the four case studies referred to throughout this chapter and the next, two are from scientific computing, one is from computer graphics, and one is from commercial computing. Besides being from different application domains, the case studies are chosen to represent a range of important behaviors found in other parallel programs as well.

The first case study simulates the motion of ocean currents by discretizing the problem on a set of regular grids and solving a system of equations on the grids. This technique is very common in scientific computing and leads to a set of very common communication patterns. The second case study represents another important form of scientific computing, in which, rather than discretizing the domain on a grid, the computational domain is represented as a large number of bodies that interact with one another and move around as a result of these interactions. These so-called n-body problems are common in many areas, such as simulating galaxies in astrophysics (our specific case study), simulating proteins and other molecules in chemistry and biology, and simulating electromagnetic interactions. As in many other areas, hierarchical algorithms for solving these problems have become very popular. Hierarchical n-body algorithms, such as the one in our case study, have also been used to solve

important problems in computer graphics and some particularly difficult types of equation systems. Unlike the first case study, this one leads to irregular, long-range, and unpredictable communication.

The third case study is from computer graphics, a very important consumer of moderate-scale multiprocessors. It traverses a three-dimensional scene with highly irregular and unpredictable access patterns and renders it into a two-dimensional image for display. The first three case studies are part of a benchmark suite (Singh, Weber, and Gupta 1992) that is widely used in architectural evaluations in the literature, so a wealth of detailed information is available about them. They will be used to illustrate architectural trade-offs in this book as well.

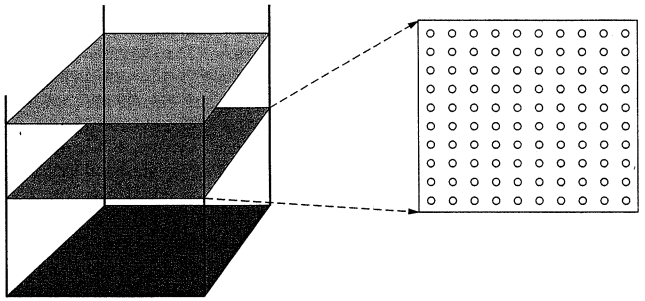
The last case study represents an increasingly important class of commercial applications that analyze the huge volumes of data being produced by our information society to discover useful knowledge, categories, and trends. These information processing applications tend to be I/O intensive, so parallelizing the I/O activity effectively is very important.

2.1.1 Simulating Ocean Currents

To model the climate of the earth, it is important to understand how the atmosphere interacts with the oceans that occupy three-fourths of the earth's surface. This case study simulates the motion of water currents in the ocean. These currents develop and evolve under the influence of several physical forces, including atmospheric effects, wind, and friction with the ocean floor. Near the ocean walls, additional "vertical" friction is present as well, which leads to the development of eddy currents. The goal of this particular application case study is to simulate these eddy currents over time and understand their interactions with the mean ocean flow.

Good models for ocean behavior are complicated: predicting the state of the ocean at any instant requires the solution of complex systems of equations, which can only be performed numerically by computer. We are, additionally, interested in the behavior of the currents over time. The actual physical problem is continuous in both space (the ocean basin) and time, but to enable computer simulation we discretize it along both dimensions. To discretize space, we model the ocean basin as a grid of points. Every important variable—such as pressure, velocity, and various currents—has a value at each grid point in this discretization. This particular application uses not a three-dimensional grid but a set of two-dimensional, horizontal cross sections through the ocean basin, each represented by a two-dimensional grid of points (see Figure 2.1). For simplicity, the ocean is modeled as a rectangular basin and the grid points are assumed to be equally spaced. Each of the many variables is therefore represented by a separate two-dimensional array for each cross section through the ocean. For the time dimension, we discretize time into a series of finite time-steps. The equations of motion are solved at all the grid points in one time-step, the state of the variables is updated as a result, the equations of motion are solved again for the next time-step, and so on repeatedly.

Every time-step itself consists of several computational phases. Many of these are used to set up values for the different variables at all the grid points using the results



(a) Cross sections (b) Spatial discretization of a cross section

FIGURE 2.1 Horizontal cross sections through an ocean basin and their spatial discretization into regular grids

from the previous time-step. In other phases, the system of equations for a time-step is actually solved. All the phases, including the solver, involve sweeping through all points of the relevant arrays and manipulating their values. The solver phases are a little more complex, as we shall see when we discuss this case study in more detail in Chapter 3.

The more grid points we use in each dimension to represent a fixed-size ocean, the finer the spatial resolution of our discretization and the more accurate our simulation. For an ocean such as the Atlantic, with its roughly 2,000 km × 2,000 km span, using a grid of 100 × 100 points implies a distance of 20 km between points in each dimension. This is not a very fine resolution, so we would like to use many more grid points. Similarly, shorter physical intervals between time-steps lead to greater simulation accuracy. For example, to simulate 5 years of ocean movement by updating the state every 8 hours, we would need about 5,500 time-steps. The computational demands for high accuracy are large, and the need for multiprocessing is clear.

Fortunately, the application naturally affords a lot of concurrency: many of the setup phases in a time-step are independent of one another and therefore can be done in parallel, and the processing of different grid points in each phase or grid computation can itself be done in parallel. For example, we might assign different parts of each ocean cross section to different processors and have the processors perform each phase of computation on their assigned parts of the cross section grids (a data parallel formulation).

2.1.2 Simulating the Evolution of Galaxies

The second case study is also from scientific computing. It seeks to understand the evolution of stars in a system of galaxies over time. For example, we may want to

study what happens when galaxies collide or how a random collection of stars folds into a defined galactic shape. This problem involves simulating the motion of a number of bodies (here stars) moving under forces exerted on each by all the others, an n-body problem. The computation is discretized in space by treating each star as a separate body or by sampling to use one body to represent many stars. Here again we discretize the computation in time and simulate the motion of the galaxies for many time-steps. In each time-step, we compute the gravitational forces exerted on each star by all the others and update the position, velocity, and other attributes of that star.

Computing the forces among stars is the most expensive part of a time-step. A simple method to compute forces is to calculate pairwise interactions among all stars. This has $O(n^2)$ computational complexity for n stars and is therefore prohibitive for the millions of stars that we would like to simulate. However, by taking advantage of insights into the force laws, smarter hierarchical algorithms are able to reduce the complexity to $O(n \log n)$. This makes it feasible to simulate problems with millions of stars in a reasonable time but only by using powerful multiprocessors. The hierarchical algorithms use the basic insight that, since the strength of the gravitational interaction falls off with distance as

$$G\frac{m_1 m_1}{r^2}$$

the influences of stars that are farther away are weaker and therefore do not need to be computed as accurately as those of stars that are close by. Thus, if a group of stars is sufficiently far from a given star, we can compute the effect of the group on the star by approximating the group as a single star at the center of the group with little loss in accuracy (see Figure 2.2). The farther away the stars are from a given star, the larger the group that can be thus approximated. In fact, the strength of many physical interactions falls off with distance, so hierarchical methods are becoming increasingly popular in many areas of computing.

The particular hierarchical force calculation algorithm used in this case study is the Barnes-Hut algorithm. (The case study is called Barnes-Hut in the literature, and this name is used here as well.) We shall see how the algorithm works in Section 3.5.2. Since galaxies are denser in some regions and sparser in others, the distribution of stars in space is highly irregular. The distribution also changes with time as the galaxy evolves. The nature of the hierarchical approach implies that stars in denser regions interact more with other stars and centers of mass—and hence have more work associated with them—than stars in sparser regions. Ample concurrency exists across stars within a time-step, but given their irregular and dynamically changing nature, the challenge is to exploit concurrency efficiently on a parallel architecture.

2.1.3 Visualizing Complex Scenes Using Ray Tracing

The third case study is the visualization of complex scenes in computer graphics. A common technique used to render such scenes into images is known as *ray tracing*.

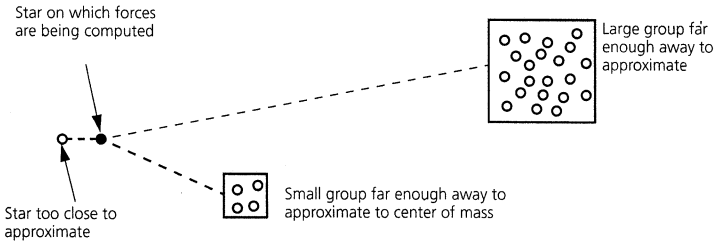


FIGURE 2.2 The insight used by hierarchical methods for *n*-body problems. A group of bodies that is far enough away from a given body may be approximated by the center of mass of the group. The farther apart the bodies, the larger the group that may be thus approximated.

The scene is represented as a set of objects in three-dimensional space, and the image being rendered is represented as a two-dimensional array of pixels (picture elements) whose color, opacity, and brightness values are to be computed. The pixels taken together represent the image, and the resolution of the image is determined by the distance between pixels in each dimension. The scene is rendered as seen from a specific viewpoint or position of the eye. Rays are shot from that viewpoint through every pixel in the image plane and into the scene. The algorithm traces the paths of these rays—computing their reflection, refraction, and lighting interactions as they strike and reflect off objects—and thus computes values for the color and brightness of the corresponding pixels. There is obvious parallelism across the rays shot through different pixels. This case study is referred to as Raytrace.

2.1.4 Mining Data for Associations

Information processing is rapidly becoming a major market for parallel systems. Businesses acquiring data about customers and products are devoting computational power to automatically extracting useful information or "knowledge" from this data. Examples from a customer database might include determining the buying patterns of demographic groups or segmenting customers according to relationships in their buying patterns. This process is called *data mining*. It differs from standard database queries in that its goal is to identify implicit trends and segmentations in data rather than simply look up the data requested by a direct, explicit query. For example, finding all customers who have bought cat food in the last week is not data mining; however, segmenting customers according to relationships in their age groups, their monthly incomes, and their preferences in cat food, in cars, and in kitchen utensils is.

A particular type of data mining is mining for associations. Here the goal is to discover relationships (associations) in the available information related to, say, different customers and their transactions and to generate rules for the inference of customer behavior. For example, the database may store for every transaction the list of items purchased in that transaction. The goal of the mining may be to determine associations between sets of commonly purchased items that tend to be purchased together—for example, the conditional probability $P(S_1/S_2)$ that a certain set of items S_1 is found in a transaction given that a different set of items S_2 is found in that transaction, where S_1 and S_2 are sets of items that occur often in customer transactions. If this probability is high, then customers who have the set S_2 in their purchase transactions may be good advertising targets of items in set S_1.

Consider the problem a little more concretely. We are given a database in which the records correspond to customer purchase transactions, as described above. Each transaction has a transaction identifier and a set of attributes, which in this case are the items purchased. The first goal in mining for associations is to examine the database and determine which sets of k items, say, are found to occur together in more than a certain threshold fraction of the transactions. A set of items (of any size) that occur together in a transaction is called an *itemset*, and an itemset that is found in more than this threshold fraction of transactions is called a *large itemset*. Once the large itemsets of size k are found, together with their frequencies of occurrence in the database, determining the association rules among them is quite easy. The problem we consider therefore focuses on discovering the large itemsets of size k and their frequencies. The database may be in main memory or more commonly on disk.

A simple way to solve the problem is to first determine the large itemsets of size one. From these, a set of candidate itemsets of size two items can be constructed—using the basic insight that an itemset can only be large if all its subsets are also large—and their frequency of occurrence in the transaction database can be counted. This results in a list of large itemsets of size two. The process is repeated until we obtain the large itemsets of size k. There is concurrency in examining large itemsets of size $k - 1$ to determine candidate itemsets of size k and in counting the number of transactions in the database that contain each of the candidate itemsets.

2.2 THE PARALLELIZATION PROCESS

The four case studies—Ocean, Barnes-Hut, Raytrace, and Data Mining—offer abundant concurrency and will help illustrate the process of creating effective parallel programs in this chapter and the next. For concreteness, we will assume that the sequential algorithm that we are to make parallel is given to us, perhaps as a description or as a sequential program. In many cases, as in these case studies, the best sequential algorithm for a problem lends itself easily to parallelization; in others, it may not afford enough parallelism, and a fundamentally different algorithm may be required. The rich field of parallel algorithm design is outside the scope of this book. However, whatever the chosen underlying sequential algorithm, a significant process of creating a good parallel program is present in all cases, and we must understand this process in order to program parallel machines effectively and evaluate architectures against parallel programs.

At a high level, the job of parallelization involves identifying the work that can be done in parallel, determining how to distribute the work and perhaps the data among the processing nodes, and managing the necessary data access, communication, and synchronization. Note that the work to be done includes computation, data access, and input/output activity. The goal is to obtain high performance while keeping programming effort and the resource requirements of the program low. In particular, we would like to obtain good speedup over the best sequential program that solves the same problem. This requires that we ensure a balanced distribution of work among processors, reduce the amount of interprocessor communication, which is expensive, and keep low the overhead of communication, synchronization, and parallelism management.

The steps in the process of creating a parallel program may be performed either by the programmer or by one of the many layers of system software that intervene between the programmer and the architecture. These layers include the compiler, the run-time system, and the operating system. In a perfect world, system software would allow users to write programs in the form they found most convenient (for example, as sequential programs in a high-level language or as an even higher-level specification of the problem) and would automatically perform the transformation into efficient parallel programs and executions. While much research is being conducted in parallelizing compiler technology and in programming languages, the goal of automatic parallelization is very ambitious and has not yet been fully achieved. In practice today, the vast majority of the process is still the responsibility of the programmer, with perhaps some help from the compiler and run-time system. Regardless of how the responsibility is divided among these parallelizing agents, the issues and trade-offs are similar, and it is important that we understand them. For concreteness, we shall assume for the most part that the programmer has to make all the decisions.

Let us now examine the parallelization process in a more structured way, by looking at the actual steps in it. Each step will address a subset of the issues needed to obtain good performance. These performance issues will be discussed in detail in Chapter 3 and only mentioned briefly here.

2.2.1 Steps in the Process

To understand the steps in creating a parallel program, let us first define three important concepts: tasks, processes, and processors. A *task* is an arbitrarily defined piece of the work done by the program. It is the smallest unit of concurrency that the parallel program can exploit; that is, an individual task is executed by only one processor, and concurrency among processors is exploited only across tasks. In the Ocean application, we can think of a single grid point in each phase of computation as being a task, or a row of grid points, or any arbitrary subset of a grid. We could even consider an entire grid computation to be a single task, in which case parallelism is exploited only across independent grid computations. In Barnes-Hut a task may be a body, in Raytrace a ray or a group of rays, and in Data Mining it may be

checking a single transaction for the occurrence of a particular itemset. What exactly constitutes a task is not prescribed by the underlying sequential program; it is a choice of the parallelizing agent, though it usually matches some natural granularity of work in the sequential program structure (for example, an iteration of a loop). If the amount of work a task performs is small, it is called a *fine-grained* task; otherwise, it is called *coarse-grained*.

A *process* (referred to interchangeably as a *thread* hereafter) is an abstract entity that performs tasks.[1] A parallel program is composed of multiple cooperating processes, each of which performs a subset of the tasks in the program. Tasks are assigned to processes by some assignment mechanism. For example, if the computation for each row in a grid in Ocean is viewed as a task, then a simple assignment mechanism may be to give an equal number of adjacent rows to each process, thus dividing the ocean cross section into as many horizontal slices as there are processes. In Data Mining, the assignment may be determined both by which portions of the database are assigned to each process and by how the candidate itemsets within a list are assigned to processes to look up the database. Processes may need to communicate and synchronize with one another to perform their assigned tasks. Finally, the way processes perform their assigned tasks is by executing them on the physical *processors* in the machine.

It is important to understand the difference between processes and processors from a parallelization perspective. While processors are physical resources, processes provide a convenient way of abstracting, or *virtualizing*, a multiprocessor: we initially write parallel programs in terms of processes, not physical processors; mapping processes to processors is a subsequent step. The number of processes does not have to be the same as the number of processors available to the program in a given execution. If there are more processes, they are multiplexed onto the available processors; if there are fewer processes, then some processors will remain idle.

Given these concepts, the job of creating a parallel program from a sequential one consists of four steps, illustrated in Figure 2.3:

1. *Decomposition* of the computation into tasks
2. *Assignment* of tasks to processes
3. *Orchestration* of the necessary data access, communication, and synchronization among processes
4. *Mapping* or binding of processes to processors

Together, decomposition and assignment are called *partitioning*, since they divide the work done by the program among the cooperating processes. Let us examine the steps and their individual goals a little further.

1. In Chapter 1 we used the correct operating systems definition of a process: an address space and one or more threads of control that share that address space. Thus, processes and threads are distinguished in that definition. To simplify our discussion of parallel programming in this chapter, we do not make this distinction but assume that a process has only one thread of control.

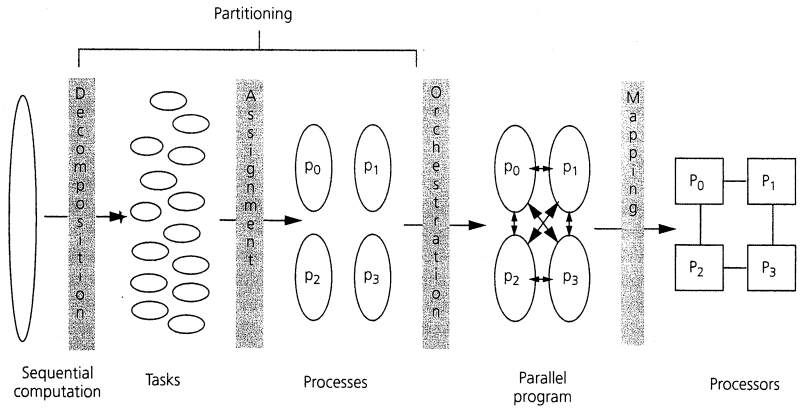


FIGURE 2.3 Steps in parallelization and the relationships among tasks, processes, and processors. The decomposition and assignment phases together are called *partitioning*. The orchestration phase coordinates data access, communication, and synchronization among processes (p), and the mapping phase maps them to physical processors (P).

Decomposition

Decomposition means breaking up the computation into a collection of tasks. In general, tasks may become available dynamically as the program executes, and the number of tasks available at a time may vary over the execution of the program. The maximum number of tasks available for execution at a time provides an upper bound on the number of processes (and hence processors) that can be used effectively at that time. Therefore, the major goal in decomposition is to expose enough concurrency to keep the processes busy at all times, yet not so much that the overhead of managing the tasks becomes substantial compared to the useful work done.

Limited concurrency is the most fundamental limitation on the speedup achievable through parallelism. It is not only the available concurrency in the underlying problem that matters but also how much of this concurrency is exposed in the decomposition. The impact of available concurrency is codified in one of the few "laws" of parallel computing, called Amdahl's Law. If some portions of a program's execution don't have as much concurrency as the number of processors used, then some processors will have to be idle for those portions and speedup will be suboptimal. To see this in its simplest form, consider what happens if a fraction s of a program's execution time on a uniprocessor is inherently sequential; that is, it cannot be parallelized. Even if the rest of the program is parallelized to run on a large number of processors in infinitesimal time, this sequential time will remain. The overall execution time of the parallel program will be at least s, normalized to a total

sequential time of 1, and the speedup limited to 1/s. For example, if $s = 0.2$ (20% of the program's execution is sequential), the maximum speedup available is 1/0.2, or 5, regardless of the number of processors used, even if we ignore all other sources of overhead. Example 2.1 provides a simple but more realistic example.

EXAMPLE 2.1 Consider an example program with two phases. In the first phase, a single operation is performed independently on all points of a two-dimensional *n*-by-*n* grid, as in Ocean. In the second phase, the sum of the n^2 grid point values is computed. If we have *p* processors, we can assign n^2/p points to each processor and complete the first phase in parallel in time n^2/p. In the second phase, each processor can add each of its assigned n^2/p values into a global sum variable. What is the problem with this assignment, and how can we expose more concurrency? Ignore the costs of data access and communication.

Answer The problem is that the accumulations into the global sum must be done one at a time, or *serialized*, to avoid corrupting the sum value by having two processors try to modify it simultaneously (see mutual exclusion in Section 2.3.5). Thus, the second phase is effectively serial and takes n^2 time regardless of *p*. The total time in parallel is $n^2/p + n^2$, compared to a sequential time of $2n^2$, so the speedup is at most

$$\frac{2n^2}{\frac{n^2}{p} + n^2}$$

or

$$\frac{2p}{p+1}$$

which is at best 2 even if a very large number of processors is used.

We can expose more concurrency by using a little trick. Instead of summing each value directly into the global sum and serializing all the summing, we divide the second phase into two phases. In the new second phase, a process sums its assigned values independently into a private sum. Then, in the third phase, processes sum their private sums into the global sum. The second phase is now fully parallel; the third phase is serialized as before, but there are only *p* operations in it, not *n*. The total parallel time is $n^2/p + n^2/p + p$, and the speedup is at best $p \times 2n^2/(2n^2 + p^2)$. If *n* is large relative to *p*, this speedup limit is almost linear in the number of processors used. Figure 2.4 illustrates the improvement and the impact of limited concurrency. ■

More generally, given a decomposition and a problem size, we can construct a *concurrency profile*, which depicts how many operations (or tasks) are available to be performed concurrently in the application at a given time. The concurrency profile is a function of the problem, the decomposition, and the problem size. However, it is independent of the number of processors, effectively assuming that an infinite number of processors is available. It is also independent of the assignment or orchestration. These concurrency profiles may be easy to provide analytically (as in Example 2.1 and as we shall see for matrix factorization in Exercise 3.8) or they may be quite irregular. For example, Figure 2.5 shows a concurrency profile of a parallel event-driven simulation for the synthesis of digital logic systems. The *x*-axis is time,

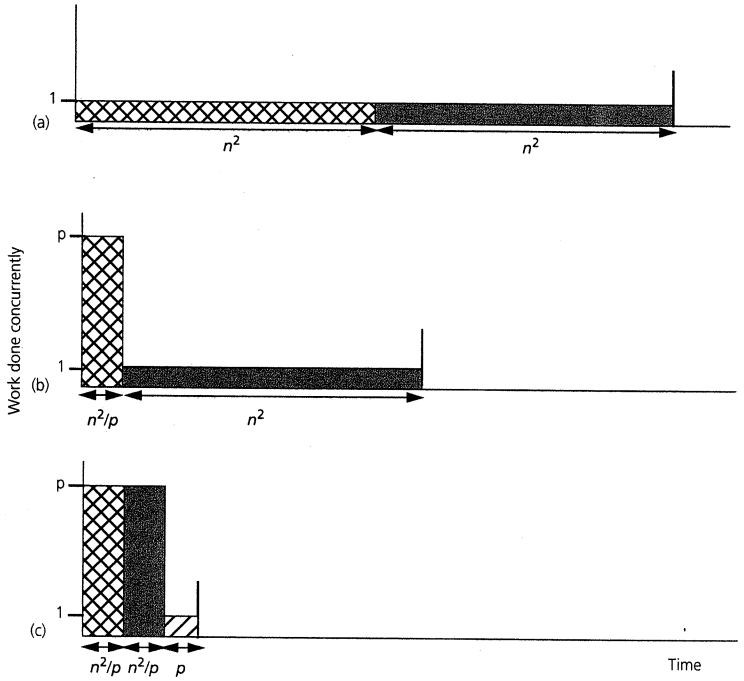


FIGURE 2.4 Illustration of the impact of limited concurrency: (a) one processor; (b) *p* processors, n^2 operations serialized; (c) *p* processors, *p* operations serialized. The *x*-axis is time, and the *y*-axis is the amount of work available (exposed by the decomposition) to be done in parallel at a given time. (a) shows the profile for a single processor. (b) shows the original case in the example, which is divided into two phases: one fully concurrent and one fully serialized. (c) shows the improved version, which is divided into three phases: the first two fully concurrent and the last fully serialized but with a lot less work in it (*O(p)* rather than *O(n)*).

measured in clock cycles of the circuit being simulated. The *y*-axis or amount of concurrency is the number of logic gates in the circuit that are ready to be evaluated at a given time, which is a function of the circuit, the values of its inputs, and time. A wide range of unpredictable concurrency exists across clock cycles, with some cycles having almost no concurrency.

The area under the curve in the concurrency profile is the total amount of work done; that is, the number of operations or tasks computed or the "time" taken on a single processor. Its horizontal extent is a lower bound on the time that it would

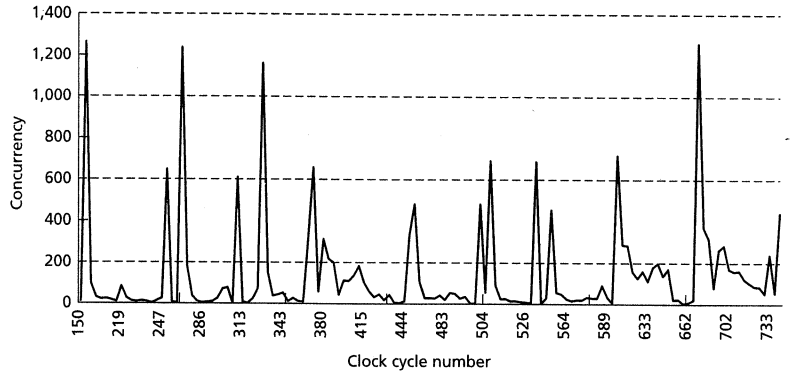


FIGURE 2.5 Concurrency profile for a distributed-time, discrete-event logic simulator. The circuit being simulated is a simple MIPS R6000 microprocessor. The *y*-axis shows the number of logic elements available for evaluation in a given simulated clock cycle.

take to run the best parallel program given that decomposition, assuming an infinitely large number of processors and that data access and communication are free. The area divided by the horizontal extent therefore gives us a limit on the achievable speedup with an unlimited number of processors, which is simply the average concurrency available in the application over time. A rewording of Amdahl's Law may therefore be

$$\text{Speedup} \leq \frac{\text{Area under Concurrency Profile}}{\text{Horizontal Extent of Concurrency Profile}}$$

For p processors, if f_k is the number of x-axis points in the concurrency profile that have concurrency k, then we can write Amdahl's Law as

$$\text{Speedup } (p) \leq \frac{\sum_{k=1}^{\infty} f_k k}{\sum_{k=1}^{\infty} f_k \left\lceil \frac{k}{p} \right\rceil} \tag{2.1}$$

It is easy to see that if the total work

$$\sum_{k=1}^{\infty} f_k k$$

is normalized to 1 and a fraction s of this is serial, then the speedup with an infinite number of processors is limited by $1/s$, and that with p processors it is limited by

$$\frac{1}{s + \frac{1-s}{p}}$$

In fact, Amdahl's Law can be applied to any overhead of parallelism (not just limited concurrency) that is not alleviated by using more processors. For now, Amdahl's Law quantifies the importance of exposing enough concurrency as a first step in creating a parallel program.

Assignment

Assignment means specifying the mechanism by which tasks will be distributed among processes. For example, which process is responsible for computing forces on which stars in Barnes-Hut? Which process will count occurrences of which itemsets, and in which parts of the database, in Data Mining?

The primary performance goals of assignment are to balance the workload among processes, to reduce the amount of interprocess communication, and to reduce the run-time overhead of managing the assignment. Balancing the workload is often referred to as *load balancing*. The workload to be balanced includes computation, input/output, and data access or communication; programs that are not balanced well among processes are said to be *load imbalanced*. Interprocess communication is expensive, especially when the processes run on different processors, and complex assignments of tasks to processes may incur overhead at run time.

Achieving these performance goals simultaneously can appear intimidating. However, most programs lend themselves to a fairly structured approach to partitioning (i.e., decomposition and assignment). For example, programs are often structured in phases, and candidate tasks for decomposition within a phase are often easily identified as seen in the case studies. The appropriate assignment of tasks is often discernible either by inspection of the code or from a higher-level understanding of the application. Where this is not so, well-known heuristic techniques are often applicable.

If the assignment is completely determined at the beginning of the program, or just after reading and analyzing the input, and does not change thereafter, it is called a *static* or *predetermined assignment*; if the assignment of work to processes is determined at run time as the program executes (perhaps to react to load imbalances), it is called a *dynamic assignment*. We shall see examples of both in Chapter 3. Note that this use of "static" is a little different from the compile-time meaning typically used in computer science. Compile-time assignment that does not change at run time would indeed be static, but the term is more general here.

Decomposition and assignment are the major algorithmic steps in parallelization. They are usually independent of the underlying architecture and programming model, although sometimes the cost and complexity of using certain primitives on a system can impact decomposition and assignment decisions. As architects, we assume that the programs that will run on our machines are reasonably partitioned. There is nothing we can do if a computation is not parallel enough or not balanced across processes and little we may be able to do if it overwhelms the machine with communication. As programmers, we usually focus on decomposition and assignment first, independent of the programming model or architecture, though in some cases the properties of the latter may cause us to revisit our partitioning strategy.

Orchestration

Orchestration is the step in which the architecture and programming model, as well as the programming language itself, play a large role. To execute their assigned tasks, processes need mechanisms to name and access data, to exchange data (communicate) with other processes, and to synchronize with one another. Orchestration uses the available mechanisms to accomplish these goals correctly and efficiently. The choices made in orchestration are much more dependent on the programming model, and on the efficiencies with which the primitives of the programming model are supported, than the choices made in the previous steps. Some questions in orchestration include how to organize data structures, how to schedule the tasks assigned to a process temporally to exploit data locality, whether to communicate implicitly or explicitly and in small or large messages, and how exactly to organize and express the interprocess communication and synchronization that resulted from assignment. The programming language is important both because this is the step in which the program is actually written and because some of the trade-offs in orchestration are influenced strongly by available language mechanisms and their costs.

The major performance goals in orchestration are reducing the cost of the communication and synchronization as seen by the processors, preserving locality of data reference, scheduling tasks so that those on which many other tasks depend are completed early, and reducing the overhead of parallelism management. The job of architects is to provide the appropriate primitives with efficiencies that simplify successful orchestration. We shall discuss the major aspects of orchestration further when we see how programs are actually written.

Mapping

The cooperating processes that result from the decomposition, assignment, and orchestration steps constitute a full-fledged parallel program on modern systems. The program may choose to control the mapping of processes to processors, but if not, the operating system will take care of it, providing a parallel execution. Mapping tends to be fairly specific to the system or programming environment.

In the simplest case, the processors in the machine are partitioned into fixed subsets, possibly the entire machine, and only a single program runs at a time in a subset. This is called *space-sharing* of the machine. The program can bind, or *pin*, processes to processors to ensure that they do not migrate during the execution; it can even control exactly which processor a process runs on so as to preserve locality of communication in the network topology. Strict space-sharing schemes, together

with some simple mechanisms for time-sharing a subset among multiple applications, have so far been typical of large-scale multiprocessors. At the other extreme, the operating system may dynamically control which process runs where and when—without allowing the user any control over the mapping—to achieve better aggregate resource sharing and utilization. Each processor may employ the usual multiprogrammed scheduling criteria to manage processes from the same or different programs, and processes may be moved around among processors as the scheduler dictates. The operating system may extend the scheduling criteria to include multiprocessor-specific issues (for example, trying to have a process be scheduled on the same processor as much as possible so that the process can reuse its state in the processor cache and trying to schedule processes from the same application at the same time). In fact, most modern systems fall somewhere between these two extremes: the user may ask the system to preserve certain properties, giving the user program some control over the mapping, but the operating system is allowed to change the mapping dynamically for effective resource management.

Mapping and associated resource management issues in multiprogrammed systems are active areas of research. However, our goal here is to understand parallel programming in its basic form, so for simplicity we assume that a single parallel program has complete control over the resources of the machine. We also assume that the number of processes equals the number of processors and that neither changes during the execution of the program. By default, the operating system will place one process on every processor in no particular order. Processes are assumed not to migrate from one processor to another during execution. For this reason, the terms "process" and "processor" are used interchangeably in the rest of the chapter.

2.2.2 Parallelizing Computation versus Data

The view of the parallelization process described above has been centered on computation, or work, rather than on data. It is the computation that is decomposed and assigned. However, due to the programming model or performance considerations, we may be responsible for decomposing and assigning data to processes as well. In fact, in many important classes of problems, the decomposition of work and data are so strongly related that they are difficult or even unnecessary to distinguish. Ocean is a good example: each cross-sectional grid through the ocean is represented as an array, and we can view the parallelization as decomposing the data in each array and assigning parts of it to processes. The process that is assigned a portion of an array will then be responsible for the computation associated with that portion; this is known as an *owner computes* arrangement. A similar situation exists in Data Mining, where we can view the database as being decomposed and assigned; of course, here is also the question of assigning the itemsets to processes. Several language systems, including the high-performance Fortran standard (Koebel et al. 1994; High Performance Fortran Forum 1993), allow the programmer to specify the decomposition and assignment of data structures. The assignment of computation then follows the assignment of data in an owner computes manner. However, the distinction between

computation and data is stronger in many other more irregular applications, including the Barnes-Hut and Raytrace case studies, as we shall see. Since the computation-centric view is more general, we shall retain this view and consider data management to be part of the orchestration step.

2.2.3 Goals of the Parallelization Process

As stated previously, the major goal of using a parallel machine is to improve performance by obtaining speedup over the best uniprocessor execution. Each of the steps in creating a parallel program has a role to play in achieving this overall goal, and each step has its own subset of performance goals. These are summarized in Table 2.1; the next chapter discusses them in more detail.

Creating an effective parallel program requires evaluating cost as well as performance. In addition to the dollar cost of the machine itself, we must consider the resource requirements of the program on the architecture (for example, its memory usage) and the effort it takes to develop a satisfactory program. While costs and their impact are often more difficult to quantify than performance, they are very important, and we must not lose sight of them; in fact, we often decide to compromise performance to reduce them. As algorithm designers, we should favor high-performance solutions that keep the resource requirements of the algorithm small and that don't require inordinate programming effort. As architects, we should try to design high-performance systems that facilitate resource-efficient algorithms and reduce programming effort in addition to being low cost. For example, an architecture on which performance improves gradually with increased programming effort may be preferable to one that is capable of ultimately delivering better performance but requires inordinate programming effort to even achieve acceptable performance.

Table 2.1 Steps in the Parallelization Process and Their Goals

Step	Architecture-Dependent?	Major Performance Goals
Decomposition	Mostly no	Expose enough concurrency but not too much
Assignment	Mostly no	Balance workload Reduce communication volume
Orchestration	Yes	Reduce noninherent communication via data locality Reduce communication and synchronization cost as seen by the processor Reduce serialization at shared resources Schedule tasks to satisfy dependences early
Mapping	Yes	Put related processes on the same processor if necessary Exploit locality in network topology

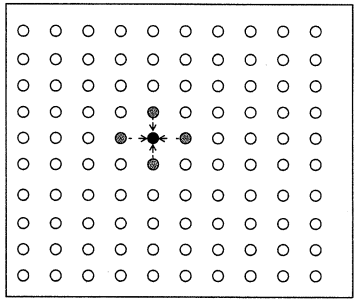
We can apply an understanding of the basic process and its goals to a simple but detailed example to see what the resulting parallel programs look like in the three major modern programming models introduced in Chapter 1: shared address space, message passing, and data parallel. Our focus will be on illustrating programs and programming primitives, not so much on performance, which is the subject of Chapter 3.

2.3 PARALLELIZATION OF AN EXAMPLE PROGRAM

The four case studies introduced at the beginning of the chapter all lead to parallel programs that are too complex and too long to serve as useful sample programs. Instead, this section presents a simplified version of a piece, or *kernel*, of Ocean: its equation solver. It uses the equation solver to dig deeper and to illustrate how to implement a parallel program using the three programming models. Except for the data parallel version, which necessarily uses a high-level data parallel language, the parallel programs are not written in an aesthetically pleasing language that relies on software layers to hide the orchestration and communication abstraction from the programmer. Rather, they are written in C or Pascal-like pseudocode augmented with simple extensions for parallelism, thus exposing the basic communication and synchronization primitives that a shared address space or message-passing communication abstraction must provide. Standard sequential languages augmented with primitives for parallelism also reflect the state of most real parallel programming today.

2.3.1 The Equation Solver Kernel

The equation solver kernel solves a simple partial differential equation on a grid, using what is referred to as a finite differencing method. It operates on a regular, two-dimensional grid or array of $(n + 2)$-by-$(n + 2)$ elements, such as a single horizontal cross section of the ocean basin in Ocean. The border rows and columns of the grid contain boundary values that do not change, whereas the interior n-by-n points are updated by the solver, starting from their initial values. The computation proceeds over a number of sweeps. In each sweep, it operates on all the interior n-by-n points of the grid. For each point, it replaces its value with a weighted average of itself and its four nearest-neighbor points—above, below, left, and right (see Figure 2.6). The updates are done in place in the grid, so the update computation for a point sees the new values of the points above and to the left of it and the old values of the points below and to its right. This form of update is called the Gauss-Seidel method. During each sweep, the kernel also computes the average difference of an updated element from its previous value. If this average difference over all elements is smaller than a predefined "tolerance" parameter, the solution is said to have converged and the solver exits at the end of the sweep. Otherwise, it performs another sweep and tests for convergence again. The sequential pseudocode is shown in Figure 2.7. Let us now go through the steps to convert this simple equation solver



Expression for updating each interior point:

$$A[i,j] = 0.2 \times (A[i,j] + A[i,j-1] + A[i-1,j] + A[i,j+1] + A[i+1,j])$$

FIGURE 2.6 Nearest-neighbor update of a grid point in the simple equation solver. The black point is $A[i,j]$ in the two-dimensional array that represents the grid and is updated using itself and the four shaded points that are its nearest neighbors according to the equation at the right of the figure.

to a parallel program for each programming model. The decomposition and assignment steps are essentially the same for all three models, so these steps are examined in a general context. Once we enter the orchestration phase, the discussion will be organized explicitly by programming model.

2.3.2 Decomposition

For programs that are structured in successive loops or loop nests, a simple way to identify concurrency is to start from the loop structure itself. We examine the individual loops or loop nests in the program one at a time, see if their iterations can be performed in parallel, and determine whether this exposes enough concurrency. We can then look for concurrency across loops or take a different approach if necessary. Let us follow this program-structure-based approach in Figure 2.7.

Each iteration of the outermost while loop, beginning at line 15, sweeps through the entire grid. These iterations clearly are not independent since data modified in one iteration is accessed in the next. Consider the loop nest in lines 17–24, and ignore the lines containing `diff`. Look at the inner loop first (the j loop starting on line 18). Each iteration of this loop reads the grid point $(A[i,j-1])$ that was written in the previous iteration. The iterations are therefore sequentially dependent, and we call this a *sequential loop*. The outer loop of this nest is also sequential, since the elements in row $i-1$ were written in the previous $(i-1\text{th})$ iteration of this loop. So this simple analysis of existing loops and their dependences uncovers no concurrency in this example program.

In general, an alternative to relying on program structure to find concurrency is to go back to the fundamental dependences in the underlying algorithms used,

```
1.  int n;                                  /*size of matrix: (n + 2-by-n + 2) elements*/
2.  float **A, diff = 0;

3.  main()
4.  begin
5.    read(n) ;                             /*read input parameter: matrix size*/
6.    A ← malloc (a 2-d array of size n + 2 by n + 2 doubles);
7.    initialize(A);                        /*initialize the matrix A somehow*/
8.    Solve (A);                            /*call the routine to solve equation*/
9.  end main

10. procedure Solve (A)                     /*solve the equation system*/
11.   float **A;                            /*A is an (n + 2)-by-(n + 2) array*/
12. begin
13.   int i, j, done = 0;
14.   float temp;
15.   while (!done) do                      /*outermost loop over sweeps*/
16.     diff = 0;                           /*initialize maximum difference to 0*/
17.     for i ← 1 to n do                   /*sweep over nonborder points of grid*/
18.       for j ← 1 to n do
19.         temp = A[i,j];                  /*save old value of element*/
20.         A[i,j] ← 0.2 * (A[i,j] + A[i,j-1] + A[i-1,j] +
21.           A[i,j+1] + A[i+1,j]);         /*compute average*/
22.         diff += abs(A[i,j] - temp);
23.       end for
24.     end for
25.     if (diff/(n*n) < TOL) then done = 1;
26.   end while
27. end procedure
```

FIGURE 2.7 Pseudocode describing the sequential equation solver kernel. The main body of work to be done in each sweep is in the nested for loop in lines 17–23. This is what we would like to parallelize. (Italics indicate keywords of the sequential programming language.)

regardless of program or loop structure. In the equation solver, we might look at the fundamental dependences in the generation and usage of data values (*data dependences*) at the granularity of individual grid points. As discussed earlier, since the computation proceeds from left to right and top to bottom in the grid, computing a particular grid point in the sequential program uses the updated values of the grid points directly above and to the left. This data dependence pattern is shown in Figure 2.8. The result is that the elements along a given anti-diagonal (southwest to northeast) have no dependences among them and can be computed in parallel, whereas the points in the next anti-diagonal depend on some points in the previous one. From this diagram, we can observe that of the $O(n^2)$ work involved in each

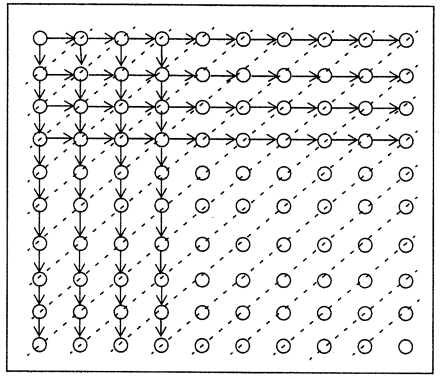


FIGURE 2.8 Dependences and concurrency in the Gauss-Seidel equation solver computation. The horizontal and vertical lines with arrows indicate dependences; the anti-diagonal, dashed lines connect points with no dependences among them that can be computed in parallel.

sweep, there is an inherent concurrency proportional to n along anti-diagonals and a sequential dependence proportional to n along the diagonal.

Suppose we decide to decompose the work into individual grid points so that updating a single grid point is a task. We can exploit the concurrency this exposes in several ways. First, we can leave the loop structure of the program as it is and insert point-to-point synchronization to ensure that the new value for a grid point has been produced in the current sweep before it is used by the points below or to its right. Thus, different loop nests (of the sequential program) and even different sweeps might be in progress simultaneously on different elements, as long as the element-level dependences are not violated. But the overhead of this synchronization at grid-point level may be too high. Second, we can change the loop structure: the first for loop (line 17) can be over anti-diagonals and the inner for loop can be over elements within an anti-diagonal. The inner loop can then be executed completely in parallel, with global synchronization between iterations of the outer for loop (to preserve dependences conservatively across anti-diagonals). Communication would be orchestrated very differently in the two cases, particularly if communication is in explicit messages. However, this approach also has problems. Global synchronization is still very frequent—once per anti-diagonal. In addition, the number of iterations in the parallel (inner) loop changes with successive outer loop iterations, as the size of the anti-diagonals changes, causing load imbalances among processes especially in the shorter anti-diagonals. Because of the frequency of synchronization, load imbalances, and programming complexity, neither of these approaches is used much on modern architectures.

The third and most common approach is based on exploiting knowledge of the problem beyond the dependences in the sequential program itself. The order in which the grid points are updated in the sequential algorithm (left to right and top to bottom) is in fact not fundamental to the Gauss-Seidel solution method; it is

simply one possible ordering that is convenient to program sequentially. Since the Gauss-Seidel method is not an exact solution method (unlike Gaussian elimination, for example) but rather iterates until convergence, we can update the grid points in a different order as long as we use updated values for grid points frequently enough.[2] One such ordering that is used often for parallel versions is called *red-black* ordering. The idea here is to separate the grid points into alternating red points and black points as on a checkerboard (see Figure 2.9) so that no red point is adjacent to a black point or vice versa. Since each point reads only its four nearest neighbors, to compute a given red point, we do not need the updated value of any other red point; we need only the updated values of the black points above it and to its left (in a standard sweep) and vice versa for computing black points. We can therefore divide a grid sweep into two phases, first computing all red points and then computing all black points. Within each phase no dependences exist among grid points, so we can compute all $n^2/2$ red points in parallel, synchronize globally, and then compute all $n^2/2$ black points in parallel. Global synchronization is conservative and can be replaced by point-to-point synchronization at the level of grid points since not all black points need to wait for all red points to be computed; but global synchronization is convenient.

Since the red-black ordering is different from our original sequential ordering, it can converge in fewer or more sweeps. It can also produce different final values for the grid points (though still within the convergence tolerance). While the red-point updates do not see updated values of any black points, the black points will see the updated values of all their red neighbors from the first phase of the current sweep, not just the ones to the left and above. Whether the new order is sequentially better or worse than the old one depends on the problem. The red-black ordering also has the advantage that the produced values and the convergence properties are independent of the number of processors used since no dependences occur within a phase. If the sequential program itself uses a red-black ordering, then parallelism does not change the results or convergence properties at all, thus making the parallel program *deterministic*.

Red-black ordering itself produces a longer kernel of code than is appropriate for this illustration of parallel programming. Let us examine a simpler but still common asynchronous method that does not separate points into red and black. This method simply ignores dependences among grid points within a sweep. Global synchronization is used between grid sweeps as in the preceding approach, but the loop structure for a process within a sweep is not changed from the top-to-bottom, left-to-right order. Instead, within a sweep a process simply updates the values of all its assigned grid points, accessing its nearest neighbors whether they have been updated in the current sweep by their assigned processes or not. When only a single process is used, this defaults to the original sequential ordering of updates. When multiple processes are used, the ordering is unpredictable; it depends on the assignment of

2. Even if we don't use updated values from the current sweep (i.e., while loop iteration) for any grid points but always use the values as they were at the end of the previous sweep, the system will still converge, only much slower. This is called Jacobi, rather than Gauss-Seidel, iteration.

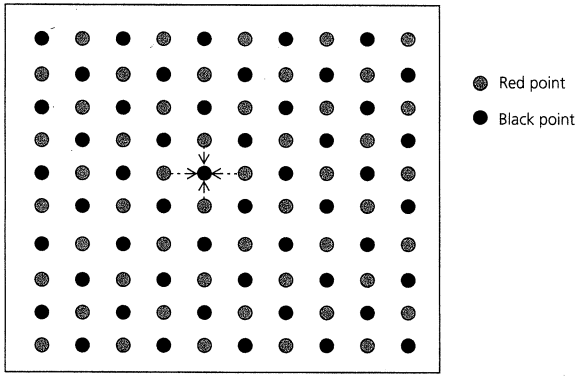


FIGURE 2.9 Red-black ordering for the equation solver. The sweep over the grid is broken up into two subsweeps: the first computes all the red points and the second all the black points. Since red points depend only on black points and vice versa, no dependences occur within a subsweep.

points to processes, the number of processes used, and how quickly different processes execute relative to one another at run time. The execution is no longer deterministic, and the number of sweeps required to converge may depend on the number of processes used; however, for most reasonable assignments the number of sweeps will not vary much.

If we choose a decomposition into individual inner loop iterations (grid points), we can express the program by revising lines 15–26 of Figure 2.7. Figure 2.10 highlights the changes to the code in boldface: all we have done is replace the keyword `for` in the parallel loops with `for_all`. A `for_all` loop simply tells the underlying hardware/software system that all iterations of the loop can be executed in parallel without worrying about dependences, but it says nothing about assignment. A loop nest with both nesting levels being `for_all` means that all iterations in the loop nest (`n*n` or n^2 here) can be executed in parallel. The system can assign and orchestrate the parallelism in any way it chooses; the program does not take a position on this. All it assumes is an implicit global synchronization after a `for_all` loop nest.

In fact, we can decompose the computation not just into individual inner loop iterations but into any aggregated groups of iterations we desire. Notice that decomposing the computation corresponds very closely to decomposing the grid itself. Suppose we wanted to decompose into rows of grid points instead so that the work for an entire row is an indivisible task that must be assigned to the same process. We could express this by making the inner loop on line 18 a sequential loop, changing its `for_all` back to a `for`, but leaving the loop over rows on line 17 as a parallel

```
15.  while (!done) do                          /*a sequential loop*/
16.    diff = 0;
17.    for_all i ← 1 to n do                   /*a parallel loop nest*/
18.      for_all j ← 1 to n do
19.        temp = A[i,j];
20.        A[i,j] ← 0.2 * (A[i,j] + A[i,j-1] + A[i-1,j] +
21.          A[i,j+1] + A[i+1,j]);
22.        diff += abs(A[i,j] - temp);
23.      end for_all
24.    end for_all
25.    if (diff/(n*n) < TOL) then done = 1;
26.  end while
```

FIGURE 2.10 Parallel equation solver kernel with decomposition into grid points and no explicit assignment. Since both for loops are made parallel by using `for_all` instead of `for`, the decomposition is into individual grid elements. Other than this change, the code is the same as the sequential code.

`for_all` loop. The parallelism, or *degree of concurrency*, exploited under this decomposition is reduced from n^2 inherent in the problem to n: instead of n^2 independent tasks of duration 1 unit each, we now have n independent tasks of duration n units each. If each task is executed on a different processor, we will have approximately $2n$ words of communication (accesses to grid points that were computed by other processors) for n points, which results in a communication-to-computation ratio of $O(1)$.

2.3.3 Assignment

Using the row-based decomposition, let us see how we might assign rows to processes explicitly. The simplest option is a static (predetermined) assignment in which each process is responsible for a contiguous block of rows, as shown in Figure 2.11. Interior row i is assigned to process

$$\left\lfloor \frac{i}{\frac{n}{p}} \right\rfloor$$

where p is the number of processes. Alternative static assignments to this so-called block assignment are also possible, such as a *cyclic* assignment in which rows are interleaved among processes (process i is assigned rows i, $i + p$, and so on). We might also consider a dynamic assignment where each process repeatedly grabs the next available (not yet computed) row after it finishes with a row task, so that it is not predetermined which process computes which rows. For now, we will work with the static block assignment. This simple partitioning of the problem exhibits good load balance across processes as long as the number of interior rows is divisible by the number of processes, since the work per row is uniform across rows. Observe

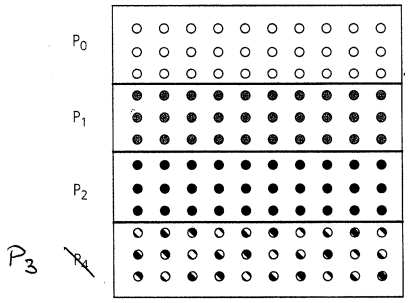


FIGURE 2.11 A simple assignment for the parallel equation solver. Each of the four processors is assigned a contiguous, equal number of rows of the grid. In each sweep, a processor will perform the work needed to update the elements of its assigned rows. Only the interior rows, which are updated in a sweep, are shown in the figure.

that the static assignments have further reduced the parallelism or degree of concurrency, from n to p, by making tasks larger, and the block assignment has reduced the communication required by assigning adjacent rows to the same processor. The communication-to-computation ratio is now only

$$O\left(\frac{p}{n}\right)$$

Having examined decomposition and assignment, we are ready to dig into the orchestration phase. This requires that we pin down the programming model. We begin with a high-level, data parallel model and then look at the two major programming models that the data parallel and other models might compile down to: shared address space and explicit message passing.

2.3.4 Orchestration under the Data Parallel Model

The data parallel model is convenient for the equation solver kernel since it is natural to view the computation as a single thread of control performing global transformations on a large array data structure (Hillis 1985; Hillis and Steele 1986). Computation and data are quite interchangeable, a simple decomposition and assignment of the data leads to good load balance across processes, and the appropriate assignments (partitions) are very regular in shape and can be described by simple expressions. Pseudocode for the data parallel equation solver is shown in Figure 2.12. We assume that global declarations (outside any procedure) describe shared data and that all other data (for example, data on a procedure's stack) is private to a process. Dynamically allocated shared data, such as the array `A`, is allocated with a `G_MALLOC` (global `malloc`) call rather than a regular `malloc`. The `G_MALLOC` allocates data in a shared region of the heap storage, which can be accessed and modified by any process. Other than this, the main differences (shown in boldface) from the sequential program are the use of `for_all` loops instead of

```
1.   int n, nprocs;                  /*grid size (n + 2-by-n + 2) and number of processes*/
2.   float **A, diff = 0;

3.   main()
4.   begin
5.     read(n); read(nprocs);; /*read input grid size and number of processes*/
6.     A ← G_MALLOC (a 2-d array of size n+2 by n+2 doubles);
7.     initialize(A);               /*initialize the matrix A somehow*/
8.     Solve (A);                   /*call the routine to solve equation*/
9.   end main

10.  procedure Solve(A)             /*solve the equation system*/
11.    float **A;                   /*A is an (n + 2-by-n + 2) array*/
12.  begin
13.    int i, j, done = 0;
14.    float mydiff = 0, temp;
14a.   DECOMP A[BLOCK,*, nprocs];
15.    while (!done) do             /*outermost loop over sweeps*/
16.      mydiff = 0;                /*initialize maximum difference to 0*/
17.      for_all i ← 1 to n do      /*sweep over non-border points of grid*/
18.        for_all j ← 1 to n do
19.          temp = A[i,j];         /*save old value of element*/
20.          A[i,j] ← 0.2 * (A[i,j] + A[i,j-1] + A[i-1,j] +
21.            A[i,j+1] + A[i+1,j]);    /*compute average*/
22.          mydiff += abs(A[i,j] - temp);
23.        end for_all
24.      end for_all
24a.     REDUCE (mydiff, diff, ADD);
25.      if (diff/(n*n) < TOL) then done = 1;
26.    end while
27.  end procedure
```

FIGURE 2.12 Pseudocode describing the data parallel equation solver. Differences from the sequential code are shown in boldface. Italicized boldface indicates constructs designed to achieve parallelism. The decomposition is still into individual elements, as indicated by the nested `for_all` loop. The assignment, indicated by the (unfortunately) labeled `DECOMP` statement, is into blocks of contiguous rows (the first, or column, dimension is partitioned into blocks, and the second, or row, dimension is not partitioned). The `REDUCE` statement sums the locally computed `mydiff`s into a global `diff` value. The while loop is still serial.

`for` loops, the use of a `DECOMP` statement, the use of a private `mydiff` variable per process, and the use of a `REDUCE` statement.

We have already seen that `for_all` loops specify that the iterations can be performed in parallel. The parallel processes other than the one executing the main thread of control are implicit in the data parallel model and are active only during these parallel loops. The `DECOMP` statement has a twofold purpose. First, it specifies the assignment of the iterations to processes (`DECOMP` is in this sense an unfortunate word choice). Here, it is a `[BLOCK, *, nprocs]` assignment, which means that the first dimension (rows) is partitioned into contiguous blocks among the `nprocs` processes, and the second dimension is not partitioned at all. Specifying `[CYCLIC, *, nprocs]` would have implied a cyclic or interleaved partitioning of rows among `nprocs` processes, specifying `[BLOCK, BLOCK, nprocs]` would have implied a 2D contiguous block partitioning, and specifying `[*, CYCLIC, nprocs]` would have implied an interleaved partitioning of columns. The second and related purpose of `DECOMP` is that it also specifies how the grid data should be distributed among memories on a distributed-memory machine. (This is restricted to be the same as the assignment of computation in most current data parallel languages, following the owner computes rule, which works well in this example.) The `mydiff` variable is used to allow each process to first independently compute the sum of the difference values for its assigned grid points. Then, the `REDUCE` statement directs the system to add all the partial `mydiff` values together into the shared `diff` variable. This increases concurrency, as was discussed in Example 2.1. The `REDUCE` operation implements a *reduction*, which is a scenario in which many processes (all, in a global reduction) perform associative operations (such as addition, taking the maximum, etc.) on the same logically shared data. Associativity implies that the order of the operations does not matter. Floating-point operations such as the ones here are, strictly speaking, not associative since the way in which rounding errors accumulate depends on the order of operations. However, the effects are small and we usually ignore them, especially in iterative calculations that are approximate anyway. The reduction operation may be implemented in a library in a manner best suited to the underlying architecture.

While the data parallel programming model is well suited to specifying partitioning and data distribution for regular computations on large arrays of data (such as the equation solver kernel or the Ocean application), the suitability does not always hold true for more irregular applications, particularly those in which the communication pattern or the distribution of work among tasks changes unpredictably with time. (For example, think of the stars in Barnes-Hut or the rays in Raytrace, where assigning equal numbers of rays to processes would lead to severe load imbalances.) Let us look at the more flexible, lower-level programming models in which processes are explicit, have their own individual threads of control, and communicate with each other when they please.

2.3.5 Orchestration under the Shared Address Space Model

In a shared address space, we can simply declare the matrix A as a single shared array—as we did in the data parallel model—and processes can reference the parts of it they need using loads and stores with exactly the same array indices as in a sequential program. Communication is generated implicitly as necessary. With explicit parallel processes, we now need mechanisms to create the processes, coordinate them through synchronization, and control the assignment of work to

Table 2.2 Key Shared Address Space Primitives

Name	Syntax	Function
CREATE	CREATE(p,proc,args)	Create p processes that start executing at procedure proc with arguments args
G_MALLOC	G_MALLOC(size)	Allocate shared data of size bytes
LOCK	LOCK(name)	Acquire mutually exclusive access
UNLOCK	UNLOCK(name)	Release mutually exclusive access
BARRIER	BARRIER(name, number)	Global synchronization among number processes: none gets past BARRIER until number have arrived
WAIT_FOR_END	WAIT_FOR_END(number)	Wait for number processes to terminate
wait for flag	while (!flag); or WAIT(flag)	Wait for flag to be set (spin or block); used for point-to-point event synchronization
set flag	flag = 1; or SIGNAL(flag)	Set flag; wakes up process that is spinning or blocked on flag, if any

processes. The primitives we use are typical of low-level programming environments such as *parmacs* (Boyle et al. 1987) and are summarized in Table 2.2.

Pseudocode for the parallel equation solver in a shared address space is shown in Figure 2.13. The special primitives for parallelism are shown in boldface. They are typically implemented as library calls or macros, each of which expands to a number of instructions that accomplishes its goal. Although the code for the Solve procedure is remarkably similar to the sequential version, let's go through it one step at a time.

A single process is first started up by the operating system to execute the program, starting from the procedure called *main*. Let's call it the main process. It reads the input, which specifies the size of the grid *A* (recall that input n denotes an (n + 2)-by-(n + 2) grid of which n-by-n points are updated by the solver). It then allocates the grid *A* as a two-dimensional array in the shared address space using the G_MALLOC call (see Section 2.3.4) and initializes the grid. For data that is not dynamically allocated on the heap, different systems make different assumptions about what is shared and what is private to a process. Let us make the same assumptions as in the earlier data parallel example. Data declared outside any procedure, such as nprocs and n in Figure 2.13, is shared. Data on a procedure's stack (such as mymin, mymax, mydiff, temp, i, and j) is private to a process that executes the procedure, as is data allocated with a regular malloc call (and data that is explicitly declared to be private, not used in this program).

Having allocated data and initialized the grid, the program is ready to start solving the system. It creates (nprocs - 1) "worker" processes, which begin executing at the procedure called Solve. The main process then also calls the Solve procedure so that all nprocs processes enter the procedure in parallel as equal partners. All created processes execute the same code image until they exit from the program and terminate. That is, we use a structured, single-program-multiple-data (SPMD) style of programming. This does not mean that they proceed in lockstep or even execute the same instructions (as in the single-instruction-multiple-data or SIMD model) since, in general, they may follow different control paths through the code. Control over the assignment of work to processes—as well as what data they access—is maintained by a few private variables that acquire different values for different processes (e.g., mymin and mymax) and by simple manipulations of loop control variables. For example, we assume that every process upon creation automatically obtains a unique process identifier (pid) between 0 and nprocs - 1 in its private address space and that it uses this pid (in lines 14a–b) to determine which rows are assigned to it. Processes synchronize through calls to synchronization primitives, which will be discussed shortly.

We assume for simplicity that the total number of interior rows n is an integer multiple of the number of processes nprocs so that every process is assigned the same number of rows. Each process calculates the indices of the first and last rows of its assigned block in the private variables mymin and mymax. It then proceeds to the actual solution loop.

The outermost while loop (line 15) is still over successive grid sweeps. Although the iterations of this loop proceed sequentially, each iteration or sweep is itself executed in parallel by all processes. The decision of whether to execute the next sweep is taken separately by each process or thread of control (by setting the done variable and computing the while(!done) condition) even though in this case each will make the same decision: the redundant work performed here is very small compared to the cost of communicating a completion flag or the diff value among the processors.

The code that performs the actual updates (lines 19–22) is essentially identical to that in the sequential program. Other than the bounds in the loop control statements, which control assignment, the only difference is that each process maintains its own private variable mydiff. As in the data parallel example, this private variable keeps track of the total difference between new and old values for only its assigned grid points. It is accumulated once into the shared diff variable at the end of the sweep, rather than adding directly into the shared variable for every grid point. In addition to the serialization and concurrency reason discussed in Section 2.2.1 (Example 2.1), all processes repeatedly modifying and reading the same shared variable cause a lot of expensive communication, so we do not want to do this once per grid point.

The interesting aspect of the rest of the program (line 25 onward) is synchronization—both mutual exclusion and event synchronization. First, the accumulations into the shared variable by different processes have to be mutually exclusive. To see why, consider the sequence of instructions that a processor executes to add its

```
1.    int n, nprocs;          /*matrix dimension and number of processors to be used*/
2a.   float **A, diff;        /*A is global (shared) array representing the grid*/
                              /*diff is global (shared) maximum difference in current
                              sweep*/
2b.   LOCKDEC(diff_lock);     /*declaration of lock to enforce mutual exclusion*/
2c.   BARDEC (bar1);          /*barrier declaration for global synchronization between
                              sweeps*/

3.  main()
4.  begin
5.      read(n); read(nprocs);  /*read input matrix size and number of processes*/
6.      A ← G_MALLOC (a two-dimensional array of size n+2 by n+2 doubles);
7.      initialize(A);          /*initialize A in an unspecified way*/
8a.     CREATE (nprocs-1, Solve, A);
8.      Solve(A);               /*main process becomes a worker too*/
8b.     WAIT_FOR_END (nprocs-1); /*wait for all child processes created to terminate*/
9.  end main

10. procedure Solve(A)
11.   float **A;                /*A is entire n+2-by-n+2 shared array,
                                as in the sequential program*/
12. begin
13.   int i,j, pid, done = 0;
14.   float temp, mydiff = 0;   /*private variables*/
14a.  int mymin = 1 + (pid * n/nprocs);  /*assume that n is exactly divisible by*/
14b.  int mymax = mymin + n/nprocs - 1   /*nprocs for simplicity here*/
15.   while (!done) do          /*loop until convergence*/
16.     mydiff = diff = 0;      /*set global diff to 0 (okay for all to do it)*/
16a.    BARRIER(bar1, nprocs);  /*ensure all reach here before anyone modifies diff*/
17.     for i ← mymin to mymax do /*for each of my rows*/
18.       for j ← 1 to n do     /*for all nonborder elements in that row*/
19.         temp = A[i,j];
20.         A[i,j] = 0.2 * (A[i,j] + A[i,j-1] + A[i-1,j] +
21.           A[i,j+1] + A[i+1,j]);
22.         mydiff += abs(A[i,j] - temp);
23.       endfor
24.     endfor
25a.    LOCK(diff_lock);        /*update global diff if necessary*/
25b.    diff += mydiff;
25c.    UNLOCK(diff_lock);
25d.    BARRIER(bar1, nprocs);  /*ensure all reach here before checking if done*/
```



```
25e.    if (diff/(n*n) < TOL) then done = 1;   /*check convergence; all get
                                               same answer*/
25f.    BARRIER(bar1, nprocs);
26.   endwhile
27. end procedure
```

FIGURE 2.13 Pseudocode describing the parallel equation solver in a shared address space. Line numbers followed by a letter denote lines that were not present in the sequential version. The numbers are chosen to match the line or control structure in the sequential code with which the new lines are most closely related. The design of the data structures does not have to change from the sequential program. Processes are created with the `CREATE` call, and the main process waits for them to terminate at the end of the program with the `WAIT_FOR_END` call. The decomposition is into rows, since the inner loop is unmodified, and the outer loop specifies the assignment of rows to processes. Barriers are used to separate sweeps (and to separate the convergence test from further modification of the global `diff` variable), and locks are used to provide mutually exclusive access to the global `diff` variable.

`mydiff` variable (maintained, say, in register `r2`) into the shared `diff` variable (i.e., to execute the source statement `diff += mydiff`):

```
load the value of diff into register r1
add the register r2 to register r1
store the value of register r1 into diff
```

Suppose the value in the variable `diff` is 0 to begin with and the value of `mydiff` in each process is 1. After two processes have executed this code, we would expect the value in `diff` to be 2. However, it may turn out to be 1 instead if the processes happen to execute their operations interleaved in the following order (shown vertically):

P_1	P_2
`r1 ← diff` *{P_1 gets 0 in its r1}*	
	`r1 ← diff` *{P_2 also gets 0}*
`r1 ← r1 + r2` *{P_1 sets its r1 to 1}*	`r1 ← r1 + r2` *{P_2 sets its r1 to 1}*
`diff ← r1` *{P_1 stores 1 into diff}*	
	`diff ← r1` *{P_2 also stores 1 into diff}*

This is not what we intended. The problem is that a process (here P_2) may be able to read the value of the logically shared `diff` between the time that another process (P_1) reads it and writes it back. To prohibit this interleaving of operations, we would like the sets of operations from different processes to execute *atomically* (i.e., to achieve mutual exclusion) with respect to one another. The set of operations we want to execute atomically is called a *critical section*: once a process starts to execute the first of its three instructions above (its critical section), no other process can execute any of the instructions in its corresponding critical section until the former

process has completed its last instruction of the critical section. The LOCK-UNLOCK pair around line 25b achieves mutual exclusion for the critical section composed of `diff += mydiff`.

A lock such as `diff_lock` can be viewed as a shared token that confers an exclusive right. Acquiring the lock through the LOCK primitive gives a process the right to execute the critical section. The process that holds the lock frees it by issuing an UNLOCK command when it has completed the critical section. At this point, the lock is free for another process to either acquire or be granted, depending on the implementation. The LOCK and UNLOCK primitives must be implemented in a way that guarantees mutual exclusion. Locks are expensive, and even a given lock can cause contention and serialization if multiple processes try to access it at the same time. Our LOCK primitive takes as its argument the name of the lock being used. Associating names with locks allows us to use different locks to protect unrelated critical sections, reducing contention and serialization.

Once a process has added its `mydiff` into the global `diff`, it waits until all processes have done so and the value contained in `diff` is indeed the total difference over all grid points. This requires global event synchronization, implemented here with a BARRIER. A barrier operation takes as an argument the name of the barrier and the number of processes involved in the synchronization, and it is issued by all those processes. When a process calls the barrier, it registers the fact that it has reached that point in the program. The process is not allowed to proceed past the barrier call until the specified number of processes participating in the barrier have issued the barrier operation. That is, the semantics of `BARRIER(name,p)` are as follows: wait until `p` processes get here and only then proceed. The need for the other two barriers in the program is discussed in Exercise 2.6.

Barriers are often used to separate distinct phases of computation in a program. For example, in the Barnes-Hut galaxy simulation we use a barrier between updating the positions of the stars at the end of one time-step and using them to compute forces at the beginning of the next one, and in Data Mining we may use a barrier between counting occurrences of candidate itemsets and using the resulting large itemsets to generate the next list of candidates. Since barriers implement all-to-all event synchronization, they are usually a conservative way of preserving dependences; usually, not all operations (or processes) after the barrier actually need to wait for all operations before the barrier to complete. More specific event synchronization between pairs or groups of processes would enable some processes to get past their synchronization operation earlier; however, from a programming viewpoint it is often more convenient to use a single barrier than to orchestrate the actual dependences through point-to-point synchronization among processes.

When point-to-point synchronization is needed, one way to orchestrate it in a shared address space is with wait and signal operations on semaphores, with which we are familiar from operating systems. A more common way in parallel programs is by using normal shared variables as flags for event synchronization, as shown in Figure 2.14. Since P_1 simply spins around in a tight while loop waiting for the flag variable to be set to 1, keeping the processor busy during this time, we call this *spin-waiting* or *busy-waiting*. Recall that in the case of a semaphore the waiting process

P_1	P_2
	`A = 1;`
	`b: flag = 1;`
`a: while (flag is 0) do nothing;`	
`print A;`	

FIGURE 2.14 Point-to-point event synchronization using flags. Suppose we want to ensure that a process P_1 does not get past a certain point (say, *a*) in the program until some other process P_2 has already reached another point (say, *b*). Assume that the variable flag (and *A*) was initialized to 0 before the processes arrived at this scenario. If P_1 gets to statement *a* after P_2 has already executed statement *b*, P_1 will simply pass point *a*. If, on the other hand, P_2 has not yet executed *b*, then P_1 will remain in the "idle" while loop until P_2 reaches *b* and sets flag to 1, at which point P_1 will exit the idle loop and proceed. If we assume that the writes by P_2 are seen by P_1 in the order in which P_2 issues them, then this synchronization will ensure that P_1 prints the value 1 for *A*.

does not spin and consume processor resources but rather blocks (suspends) itself and is awakened when another process signals the semaphore.

In event synchronization among subsets of processes, or *group event synchronization*, one or more processes may wait for an event and one or more processes may notify them of its occurrence. Group event synchronization can be orchestrated either using ordinary shared variables as flags or by using barriers among subsets of processes.

Returning to the equation solver in Figure 2.13, once a process is past the barrier, it reads the value of `diff` and examines whether the average difference over all grid points (`diff/(n*n)`) is less than the error tolerance used to determine convergence. If so, it sets the `done` flag to exit from the while loop; if not, it goes on to perform another sweep.

Finally, the WAIT_FOR_END called by the main process at the end of the program (line 8b) is a particular form of all-to-one synchronization. Through it, the main process waits for all the worker processes it created to terminate. The other processes do not call WAIT_FOR_END but implicitly participate in the synchronization by terminating when they exit the `Solve` procedure that was their entry point into the program.

In summary, for this simple equation solver the parallel program in a shared address space is not too different in structure from the sequential program. The major differences in the control flow are implemented by changing the bounds on some loops. Additional differences are due to creating processes, partitioning the work among them, and synchronizing through the use of simple and generic primitives. The body of the computational loop is mostly unchanged, as are the major data structures and the references to them. Given a strategy for decomposition, assignment, and synchronization, inserting the necessary primitives and making the necessary modifications to produce a correct parallel program is quite mechanical in this example. Changes to decomposition and assignment are also easy to incorporate

```
17.  for i ← pid+1 to n by nprocs do      /*for my interleaved set of rows*/
18.    for j ← 1 to n do                  /*for all elements in that row*/
19.      temp = A[i,j];
20.      A[i,j] = 0.2 * (A[i,j] + A[i,j-1] + A[i-1,j] +
21.        A[i,j+1] + A[i+1,j]);
22.      mydiff += abs(A[i,j] - temp);
23.    endfor
24.  endfor
```

FIGURE 2.15 Cyclic assignment of row-based solver in a shared address space. All that changes in the code from the block assignment of rows in Figure 2.13 is the first for statement in line 17. The data structures or accesses to them do not have to be changed.

as shown in Example 2.2. Although many simple programs have these properties in a shared address space, we will see later that more substantial changes are needed as we seek to obtain higher parallel performance and as we address more complex parallel programs.

EXAMPLE 2.2 How would the code for the shared address space parallel version of the equation solver (Figure 2.13) change if we retained the same decomposition into rows but changed to a cyclic (interleaved) assignment of rows to processes?

Answer Figure 2.15 shows the relevant pseudocode. All that has changed in the code is the control arithmetic in line 17. The same global data structure is used with the same indexing, and the rest of the parallel program stays exactly the same. ■

2.3.6 Orchestration under the Message-Passing Model

We now examine a possible implementation of the parallel solver using explicit message passing between private address spaces, employing the same decomposition and assignment as before. Since we no longer have a shared address space, we cannot simply declare the matrix A to be shared and have processes reference parts of it as they would in a sequential program. Rather, the logical data structure A must be represented by a collection of smaller per-process data structures, which are allocated among the private address spaces of the cooperating processes in accordance with the assignment of work. In particular, the process that is assigned a block of rows allocates those rows as an array in its private address space.

A set of simple primitives for message-passing programming are shown in Table 2.3. The message-passing program shown in Figure 2.16 uses some of these primitives and is structurally very similar to the shared address space program in Figure 2.13 (more complex programs will reveal further differences in Section 3.6). Here too a main process is started by the operating system when the program executable is invoked, and this main process creates `nprocs - 1` other processes to collaborate with it. We assume again that every created process automatically acquires a

Table 2.3 Some Basic Message-Passing Primitives

Name	Syntax	Function
CREATE	CREATE(procedure)	Create process that starts at procedure
SEND	SEND(src_addr, size, dest, tag)	Send size bytes starting at src_addr to the dest process, with tag identifier
RECEIVE	RECEIVE(buffer_addr, size, src, tag)	Receive a message with the tag identifier from the src process, and put size bytes of it into buffer starting at buffer_addr
SEND_PROBE	SEND_PROBE(tag, dest)	Check if message with identifier tag has been sent to process dest (only for asynchronous message passing, and meaning depends on semantics, as discussed in this section)
RECV_PROBE	RECV_PROBE(tag, src)	Check if message with identifier tag has been received from process src (only for asynchronous message passing, and meaning depends on semantics)
BARRIER	BARRIER(name, number)	Global synchronization among number processes: none gets past BARRIER until number have arrived
WAIT_FOR_END	WAIT_FOR_END(number)	Wait for number processes to terminate

process identifier (`pid`) between 0 and `nprocs - 1`, and that the `CREATE` call automatically communicates the program's input parameters (`n` and `nprocs`) to the address space of each process.[3] The outermost loop of the `Solve` routine (line 15) still iterates over grid sweeps until convergence, and in every iteration, a process performs the computation for its assigned rows and communicates as necessary. The major differences are in orchestration: in the data structures used to represent the logically shared matrix A and in how interprocess communication and synchronization are implemented. We shall focus on these differences.

Instead of representing the matrix to be factored as a single global $(n + 2)$-by-$(n + 2)$ array A, each process in the message-passing program allocates an array called `myA` of size `(nprocs/n + 2)`-by-`(n + 2)` in its private address space. This array represents its assigned `nprocs/n` rows of the logically shared matrix A, plus two rows at the edges to hold the boundary data from its neighboring partitions (for use in the grid point updates). The boundary rows from its neighbors must be communicated to it explicitly and copied into these extra, or *ghost*, rows since their elements cannot be directly referenced otherwise as they are not in the process's

3. An alternative organization is to use what is called a "hostless" model, in which there is no single main process. The number of processes to be used is specified to the system when the program is invoked. The system then starts up that many processes and distributes the code to the relevant processing nodes. There is no need for a `CREATE` primitive in the program itself; every process reads the program inputs (`n` and `nprocs`) separately, though processes still acquire unique user-level `pids`.

```
1.   int pid, n, nprocs;              /*process id, matrix dimension and number of
                                       processors to be used*/
2.   float **myA;
3.   main()
4.   begin
5.     read(n);  read(nprocs);       /*read input matrix size and number of processes*/
8a.    CREATE (nprocs-1, Solve);
8b.    Solve();                      /*main process becomes a worker too*/
8c.    WAIT_FOR_END (nprocs-1);      /*wait for all child processes created to terminate*/
9.   end main

10.  procedure Solve()
11.  begin
13.    int i,j, pid, n' = n/nprocs, done = 0;
14.    float temp, tempdiff, mydiff = 0;     /*private variables*/
6.   myA ← malloc(a 2-d array of size [n/nprocs + 2] by n+2);
                                       /*my assigned rows of A*/
7.   initialize(myA);                  /*initialize my rows of A, in an unspecified way*/

15.  while (!done) do
16.    mydiff = 0;                    /*set local diff to 0*/
16a.   if (pid != 0) then SEND(&myA[1,0],n*sizeof(float),pid-1,ROW);
16b.   if (pid != nprocs-1) then
           SEND(&myA[n',0],n*sizeof(float),pid+1,ROW);
16c.   if (pid != 0) then RECEIVE(&myA[0,0],n*sizeof(float),pid-1,ROW);
16d.   if (pid != nprocs-1) then
           RECEIVE(&myA[n'+1,0],n*sizeof(float), pid+1,ROW);
                                       /*border rows of neighbors have now been copied
                                         into myA[0,*] and myA[n'+1,*]*/
17.    for i ← 1 to n' do              /*for each of my (nonghost) rows*/
18.      for j ← 1 to n do             /*for all nonborder elements in that row*/
19.        temp = myA[i,j];
20.        myA[i,j] = 0.2 * (myA[i,j] + myA[i,j-1] + myA[i-1,j] +
21.          myA[i,j+1] + myA[i+1,j]);
22.        mydiff += abs(myA[i,j] - temp);
23.      endfor
24.    endfor
                                       /*communicate local diff values and determine if
                                         done; can be replaced by reduction and broadcast*/
```



```
25a.   if (pid != 0) then              /*process 0 holds global total diff*/
25b.     SEND(mydiff,sizeof(float),0,DIFF);
25c.     RECEIVE(done,sizeof(int),0,DONE);
25d.   else                            /*pid 0 does this*/
25e.     for i ← 1 to nprocs-1 do      /*for each other process*/
25f.       RECEIVE(tempdiff,sizeof(float),*,DIFF);
25g.       mydiff += tempdiff;         /*accumulate into total*/
25h.     endfor
25i      if (mydiff/(n*n) < TOL) then  done = 1;
25j.     for i ← 1 to nprocs-1 do      /*for each other process*/
25k.       SEND(done,sizeof(int),i,DONE);
25l.     endfor
25m.   endif
26.  endwhile
27.  end procedure
```

FIGURE 2.16 Pseudocode describing parallel equation solver with explicit message passing. Now the meaning of the data structures and the indexing of them changes in going to the parallel code. Each process has its own myA data structure that represents its assigned part of the grid, and myA[i,j] referenced by different processes refers to different parts of the logical overall grid. The communication is all contained in lines 16a–16d and 25a–25f. No locks or barriers are needed since the synchronization is implicit in the send/receive pairs. Several extra lines of code are added to orchestrate the communication with simple sends and receives.

address space. Ghost rows are used because without them the communicated data would have to be received into separate one-dimensional arrays with different names created specially for this purpose, which would complicate the referencing of the data when they are read in the inner loop (lines 20–21). Since communicated data has to be copied into the receiver's private address space anyway, programming is made easier by extending the existing data structure rather than allocating new ones.

Recall from Chapter 1 that both communication and synchronization in a message-passing program are based on two primitives: SEND and RECEIVE. The program event that initiates data transfer is the SEND operation, unlike in a shared address space where data transfer is usually initiated by the consumer or receiver using a read (load) instruction. When a message arrives at the destination processor, it is either kept in the network queue or temporarily stored in a system buffer until a process running on the destination processor posts a RECEIVE for it. With a RECEIVE, a process reads an incoming message from the network or system buffer into a specified portion of the private (application) address space. A RECEIVE does not in itself cause any data to be transferred across the network.

The simple SEND and RECEIVE primitives used in the example program assume that the data being transferred is in a contiguous region of the virtual address space. The arguments in our simple SEND call are: the start address of the data to be sent,

which is in the sending process's private address space; the size of the message in bytes; the pid of the destination process, which we must be able to name explicitly now (unlike in a shared address space); and an optional tag or type associated with the message for matching at the receiver. The arguments to the RECEIVE call are a local address at which to place the received data, the size of the message, the sender's pid, and the optional message tag or type. The specified sender's pid and the tag, if present, are used to perform a match with the messages that have arrived and are in the system buffer, to see which one corresponds to the receive. Either or both of these fields may be wild cards, in which case they will match a message from any source process or with any tag, respectively. SEND and RECEIVE primitives are usually implemented in a library on a specific architecture, just like BARRIER and LOCK in a shared address space. A full set of message-passing primitives commonly used in real programs is part of a standard called the Message Passing Interface, or MPI (described at different levels of detail in Pacheco 1996; MPI Forum 1993; Gropp, Lusk, and Skjellum 1994). A significant extension is transfers of noncontiguous regions of memory, either with regular stride—such as every tenth word between addresses *a* and *b*, or four words every sixth word—or by using index arrays to specify unstructured addresses from which to gather data on the sending side or to which to scatter data on the receiving side. Another is a large degree of flexibility in specifying tags to match messages and in the potential complexity of a match. For example, processes may be divided into groups that communicate certain types of messages only within their group, and collective communication operations may be provided as described in the following.

Semantically, the simplest forms of SEND and RECEIVE we can use in our program are the so-called synchronous forms. A synchronous SEND operation returns control to the calling process only when it is clear that the corresponding RECEIVE has been performed. A synchronous RECEIVE returns control when the data has been received into the destination process's address space. With synchronous messages, our implementation of the communication in lines 16a–16d is actually deadlocked. All the processes issue their SEND first and stall until the corresponding receive is performed, so none will ever get to actually perform their RECEIVE! In general, synchronous message passing can easily deadlock on pairwise exchanges of data if we are not careful. One way to avoid this problem is to have every alternate process do its SENDs first followed by its RECEIVEs, and the others do their RECEIVEs first followed by their SENDs. The alternative is to use different semantic flavors of send and receive, as we shall see shortly.

The communication is done all at once at the beginning of each iteration, rather than grid point by grid point as needed in a shared address space. It could be done grid point by grid point, but the overhead of send and receive operations is usually too large to make this approach perform reasonably. As a result, unlike in the shared address space version, the message-passing program is deterministic. Even though one process updates its boundary rows while its neighbor is computing in the same sweep, the neighbor is guaranteed not to see the updates in the current sweep since they are not in its address space. A process therefore sees, in its neighbors' boundary rows, the values as they were at the end of the previous sweep, which may cause more sweeps to be needed for convergence as per our earlier discussion (red-black ordering would have been particularly useful here).

Once a process has received its neighbors' boundary rows into its ghost rows, it can update its assigned points using code almost exactly like that in the sequential and shared address space programs. (Although we use a different name, myA, for a process's local array than the A used in the sequential and shared address space programs, this is just to distinguish it from the logically shared entire grid A, which here is only conceptual; we could just as well have used the name A.) The loop bounds are different, extending from 1 to nprocs/n (substituted by n' in the code) for all processes rather than 0 to n - 1 as in the sequential program or mymin to mymax as in the shared address space program. In fact, the indices used to reference myA are local indices, which are different than the global indices that would be used if the entire logically shared grid A could be referenced as a single shared array. For example, a reference, myA[1,1], by different processes refers to different rows of the logically shared grid A. The use of local index spaces can be somewhat trickier in cases where a global index must also be used explicitly, as seen in Exercise 2.7.

Synchronization, including the accumulation of private mydiff variables into a logically shared diff variable and the evaluation of the done condition that follows, is performed very differently here than in a shared address space. Given our simple synchronous sends and receives that block the issuing process until they complete, the send/receive match encapsulates a synchronization event and no special operations (like locks and barriers) or additional variables are needed to orchestrate synchronization. Consider mutual exclusion. The logically shared diff variable must be allocated in some process's private address space (here process 0). The identity of this process must be known to all the others. Every process sends its mydiff value to process 0, which receives them all and adds them to the logically shared global diff. Since only process 0 can manipulate this logically shared variable, mutual exclusion and serialization occur naturally and no locks are needed. In fact, process 0 can simply use its own mydiff variable as the global diff.

Now consider the global event synchronization needed for determining the done condition. Once process 0 has received the mydiff values from all the other processes and accumulated them, it tests the done condition and then sends the done variable to all the other processes, which are waiting for it with receive calls. There is no need for a barrier because the completion of the synchronous receive implies that process 0 has sent out the done result and therefore that all processes' mydiffs have been accumulated. The processes then test the done condition locally to determine whether or not to proceed with another sweep. We could also, of course, implement lock and barrier calls using messages if that is more convenient for programming, although that may lead to request-reply communication and therefore more round-trip messages. More complex send/receive semantics than the synchronous ones we have used here may require additional synchronization beyond the messages themselves, as we shall see.

Notice that the code for the accumulation and done condition evaluation has expanded to several lines when using point-to-point sends and receives as communication operations. In practice, programming environments would provide library

functions like REDUCE (accumulate values from private variables in multiple processes to a single variable in a given process) and BROADCAST (send from one process to all processes) to the programmer, which the application processes could use directly to simplify the code in these stylized situations. Using these functions, lines 25a–25m in Figure 2.16 could be replaced by the five lines in Figure 2.17. The system may provide special support to improve the performance of these and other collective communication operations (such as multicast from one-to-several or even several-to-several processes, or all-to-all communication in which every process transfers data to every other process), for example, by reducing the software overhead at the sender to that of a single message, or these operations may be built on top of the usual point-to-point send and receive in user-level libraries for programming convenience only.

Finally, it was mentioned earlier that SEND and RECEIVE operations come in different semantic flavors, which we can use to solve our deadlock problem. Let us examine this a little further. The main axis along which these flavors differ is their completion semantics—that is, when they return control to the user process that issued the send or receive. These semantics affect when the data structures or buffers they operate on can be reused without compromising correctness. The two major kinds of SEND/RECEIVE are synchronous and asynchronous; within the asynchronous class are two types: blocking and nonblocking. Let us examine these options and see how they might be used in our program.

Synchronous SENDs and RECEIVEs are what we have assumed previously because they have the simplest semantics for a programmer. A synchronous SEND returns control to the calling process only when the corresponding synchronous RECEIVE at the destination end has completed successfully and returned an acknowledgment to the sender. Until the acknowledgment is received, the sending process cannot execute any code that follows the SEND. Receipt of the acknowledgment implies that the receiver has retrieved the entire message from the system buffer into the applica-

```
/*communicate local diff values and determine if done, using reduction and broadcast*/
25b.  REDUCE(0,mydiff,sizeof(float),ADD);
25c.  if (pid == 0) then
25i.     if (mydiff/(n*n) < TOL) then done = 1;
25k.  endif;
25m.  BROADCAST(0,done,sizeof(int),DONE);
```

FIGURE 2.17 Accumulation and convergence determination in the solver using REDUCE and BROADCAST instead of SEND and RECEIVE. The first argument to the REDUCE call is the destination process. All other processes will do a send to this process in the implementation of REDUCE while this process will do a receive. The next argument is the private variable to be reduced from (in all processes other than the destination) and to (in the destination process), and the third argument is the size of this variable. The last argument is the function to be performed on the variables in the reduction. Similarly, the first argument of the BROADCAST call is the sender; this process does a send and all others do a receive. The second argument is the variable to be broadcast and received into, and the third is its size. The final argument is the optional message type.

tion space. Thus, the completion of the SEND guarantees (barring hardware errors) that the message has been successfully received and that all associated data structures and buffers can be reused.

A *blocking asynchronous* (or simply *blocking*) SEND returns control to the calling process when the message has been taken from the sending application's source data structure and is therefore in the care of the system. This means that when control is returned, the sending process can modify the source data structure without affecting that message. Compared to a synchronous SEND, this allows the sending process to resume much sooner, but the return of control does not guarantee that the message has been or will actually be delivered to the appropriate process. Obtaining such a guarantee would require additional handshaking between the processes. A blocking asynchronous RECEIVE is similar to a synchronous RECEIVE in that it returns control to the calling process only when the data it is receiving has been successfully removed from the system buffer and placed at the designated application address. Once it returns, the application can immediately use the data in the specified application buffer. Unlike a synchronous RECEIVE, however, a blocking RECEIVE does not send an acknowledgment to the sender.

The *nonblocking asynchronous* (or simply *nonblocking*) SEND and RECEIVE allow the greatest overlap between computation and message passing by returning control most quickly to the calling process. A nonblocking SEND returns control immediately. A nonblocking RECEIVE returns control after simply posting the intent to RECEIVE; the actual receipt of the message and placement into a specified application data structure is performed asynchronously at an undetermined time by the system on the basis of the posted receive. In both the nonblocking SEND and RECEIVE, however, the return of control does not imply anything about the state of the message or the application data structures it uses, so it is the user's responsibility to determine that state when necessary. Separate primitives are provided to probe (query) the state. Nonblocking messages are thus typically used in a two-phase manner: first the SEND/RECEIVE operation itself and then, when needed, the probes. The probes, which must be provided by the message-passing library, might either block until the desired state is observed or might return control immediately and simply report what state was observed.

The kind of SEND/RECEIVE semantics we choose depends on how the program uses its data structures and to what degree we wish to trade off ease of programming and portability to systems with other semantics for performance. The semantics mostly affects event synchronization, since mutual exclusion falls out naturally from having only private address spaces. In the equation solver example, using asynchronous SENDs and blocking asynchronous RECEIVEs would avoid the deadlock problem since processes would proceed past the SEND and to the RECEIVE. However, if we used nonblocking asynchronous RECEIVEs, we would have to use a probe before actually using the data structure specified in the RECEIVE. Note that a blocking SEND/RECEIVE is equivalent to a nonblocking SEND/RECEIVE followed immediately by a blocking probe.

To better appreciate the differences between the shared address space and message-passing programming models, it will be instructive to perform an exercise to transform the message-passing version of the equation solver to use a cyclic assignment as we did for the shared address space version in Example 2.2. The point to observe in this case is that, although the two message-passing versions will look syntactically similar, the meaning of the `myA` data structure will be completely different. In one case it is a contiguous section of the global array, and in the other it is a set of widely separated rows. Only by careful inspection of the data structures and communication patterns can you determine how a given message-passing version corresponds to the original sequential program or its shared address space counterpart.

2.4 CONCLUDING REMARKS

The process of parallelizing a sequential application is quite structured: we decompose the work into tasks; assign the tasks to processes; orchestrate data access, communication, and synchronization among processes; and optionally map processes to processors. For many applications, including the simple equation solver used in this chapter, the initial decomposition and assignment are similar or even identical regardless of whether a shared address space or message-passing programming model is used. The differences are in orchestration, particularly in the way data structures are organized and accessed and the way communication and synchronization are performed. A shared address space allows us to use the same major data structures as in a sequential program to produce a correct parallel program. Communication is implicit through data accesses, and the decomposition of data is not required at least for correctness. In the message-passing case, we must synthesize the logically shared data structure from per-process private data structures. Communication is explicit, decomposition of data explicitly among private address spaces (processes) is necessary, and processes must be able to name one another to communicate. On the other hand, whereas a shared address space program requires additional synchronization primitives separate from the reads and writes used for implicit communication, synchronization is bundled into the explicit send and receive communication in many forms of message passing. As we examine the parallelization of more complex applications, such as the four case studies introduced in this chapter, we will understand the implications of these differences for ease of programming as well as the additional considerations imposed by the desire for high performance.

The parallel versions of the simple equation solver described here were designed to illustrate programming primitives. Although these versions will not perform terribly (e.g., we reduced communication by using a block rather than cyclic assignment of rows, and we reduced both communication and synchronization dramatically by first accumulating into local `mydiffs` and only then into a global `diff`), the programs can be improved. We shall see how in the next chapter, as we turn our attention to the performance issues in parallel programming and how positions taken on these issues affect the workload presented to the architecture.

2.5 EXERCISES

2.1 Describe two examples where a good parallel algorithm must be based on a serial algorithm that is different from the best serial algorithm since the latter does not afford enough concurrency.

2.2 Which of our case study applications (Ocean, Barnes-Hut, Raytrace, and Data Mining) do you think are amenable to decomposing data rather than computation and using an owner computes rule in parallelization? What do you think the problem(s) would be with using a strict data distribution and owner computes rule in the others?

2.3 There are two dominant models for how parent and children processes relate to each other in a shared address space. In the heavyweight so-called process model, when a process creates another process, the child gets a private copy of the parent's image; that is, if the parent had allocated a variable x, then the child also finds a variable x in its address space that is initialized to the value that the parent had for x when it created the child. However, any modifications that either process makes subsequently are to its own copy of x and are not visible to the other process. In the lightweight threads model, the child process or thread gets a pointer to the parent's image, so that it and the parent now see the same storage location for x. All data that any process or thread allocates is shared in this model, except that on a procedure's stack.

a. Consider the problem of a process having to reference its process identifier `pid` in various parts of a program, in different routines called (in a call chain) by the routine at which the process begins execution. How would you implement this in the first model? In the second? Do you need private data per process, or could you do this with all data being globally shared?

b. A program written in the former (process) model may rely on the fact that a child process gets its own private copies of the parents' data structures. What changes would you make to port the program to the latter (threads) model for data structures that are (i) only read by processes after the creation of the child and (ii) are both read and written?

2.4 The classic bounded buffer problem provides an example of point-to-point event synchronization. Two processes communicate through a finite buffer. One process, the producer, adds data items to a buffer when it is not full; another, the consumer, reads data items from the buffer when it is not empty. If the consumer finds the buffer empty, it must wait until the producer inserts an item. When the producer is ready to insert an item, it checks to see if the buffer is full, in which case it must wait until the consumer removes something from the buffer. If the buffer is empty when the producer tries to add an item, then depending on the implementation the consumer may be waiting for notification, so the producer may need to notify the consumer. Can you implement a bounded buffer with only point-to-point event synchronization, or do you need mutual exclusion as well? Design an implementation, including pseudocode.

2.5 Would you use spinning on a flag or blocking of processes for interprocess synchronization in uniprocessor operating systems? What do you think the trade-offs are between blocking and spinning on a multiprocessor?

2.6 In the shared address space parallel equation solver (Figure 2.13), why do we need the first and third barriers in a while loop iteration (lines 16a and 25f)? Can you eliminate them without inserting any other synchronization, perhaps altering when certain operations are performed? Think about all possible scenarios.

2.7 Gaussian elimination is a well-known technique for solving simultaneous linear systems of equations. Variables are eliminated one by one until there is only one left, and then the discovered values of variables are back-substituted to obtain the values of other variables. In practice, the coefficients of the unknowns in the equation system are represented as a matrix A, and the matrix is first converted to an upper-triangular matrix (a matrix in which all elements below the main diagonal are 0). Then back-substitution is used. Let us focus on the conversion to an upper-triangular matrix by successive variable elimination. Pseudocode for sequential Gaussian elimination is shown in Figure 2.18. The diagonal element for a particular iteration of the k loop is called the *pivot element*, and its row is called the *pivot row.*

a. Draw a simple figure illustrating the dependences among matrix elements.

b. Assuming a decomposition into rows and an assignment into blocks of contiguous rows, write a shared address space parallel version using the primitives used for the equation solver in this chapter.

c. Write a message-passing version for the same decomposition and assignment, first using synchronous message passing and then any form of asynchronous message passing.

d. Can you see obvious performance problems with this partitioning? (We will discuss this further in the next chapter.)

e. Modify both the shared address space and message-passing versions to use an interleaved assignment of rows to processes.

f. Discuss the trade-offs (programming difficulty and any likely major performance differences) in programming the shared address space and message-passing versions.

2.8 Suppose that a system supporting a shared address space did not support barriers but only semaphores. Even global event synchronization would have to be constructed through semaphores or ordinary flags. The use of semaphores can be illustrated as follows. Suppose process P_2 has to indicate to process P_1 (using semaphores) that P_2 has reached a point b in the program so that P_1 can proceed past a point a (where it was waiting). P_1 performs a wait (also called P or down) operation on a semaphore when it reaches point a, and P_2 performs a signal (or V or up) operation on the same semaphore when it reaches point b. If P_1 gets to a before P_2 gets to b, P_1 suspends or blocks itself and is awakened by P_2's signal operation.

a. How might you orchestrate the synchronization in the shared address space parallel Gaussian elimination with (i) flags and (ii) semaphores replacing the barriers? Could you use point-to-point or group event synchronization instead of global event synchronization?

b. Answer the same for the equation solver example.

2.9 In the straightforward, loop-based approach to parallelizing Gaussian elimination discussed so far, parallelism is exploited only within an iteration of the outermost, k, loop. Since the pivot element and its row (called the *pivot row*) are effectively broadcast directly to all processes that need it, this is called the *broadcast version.* Gaussian elimination can also be parallelized in a form that is more aggressive in exploiting the available concurrency, even across outer loop iterations. During the kth iteration, the process assigned the pivot row can simply pass the pivot row on to the next process instead of broadcasting it. This process can use the pivot row to update its assigned rows immediately, as well as pass it on to the next process, and so on. As soon as this process has done its computation for the kth iteration of that loop in the sequential program, it can immediately perform its pivot row computation for the $(k + 1)$th iteration without waiting for all other processes to receive the kth row and perform their work for the kth iteration. It can then pass this $(k + 1)$th row on to the next process as well, which can use it right away instead of waiting for the entire previous k loop iteration to complete. Multiple k loop iterations are in progress at once; rows are passed down the processor pipeline as soon as they are computed and are computed as soon as the rows needed have arrived through the pipeline. We call this the *pipelined* form of parallelization.

a. Write a shared address space pseudocode, at a similar level of detail as Figure 2.13, for a version that implements pipelined parallelism at the granularity of individual elements. Show all synchronization necessary. Do you need barriers?


```
procedure Eliminate (A)          /*triangularize the matrix A*/
begin
for k ← 0 to n-1 do              /*loop over all diagonal (pivot) elements*/
  begin
    for j ← k+1 to n-1 do        /*for all elements in the row of, and to the right of,
                                   the pivot element*/
      A[k,j] = A[k,j] / A[k,k];  /*divide by pivot element*/
    A[k,k] = 1;
    for i ← k+1 to n-1 do        /*for all rows below the pivot row*/
      for j ← k+1 to n-1 do      /*for all elements in the row*/
        A[i,j] = A[i,j] - A[i,k]* A[k,j];
      endfor
      A[i,k] = 0;
    endfor
  endfor
end procedure
```

FIGURE 2.18 Pseudocode describing sequential Gaussian elimination

b. Write a message-passing pseudocode at the level of detail of Figure 2.16 for the pipelined case in part (a). Assume that the only communication primitives you have are synchronous and asynchronous (blocking and nonblocking) sends and receives. Which versions of send and receive would you use, and why wouldn't you choose the others?

c. Discuss the trade-offs in programming the loop-based versus pipelined parallelism.

2.10 Multicast (sending a message from one process to a named list of other processes) is a useful mechanism for communicating among subsets of processes.

a. How would you implement the message-passing, interleaved assignment version of Gaussian elimination with multicast rather than broadcast? Make up a multicast primitive, write pseudocode, and compare the programming ease of the two versions.

b. Which do you think will perform better and why?

c. What group communication primitives other than multicast do you think might be useful for a message-passing system to support? Give examples of computations in which they might be used.

3

Programming for Performance

The goal of using multiprocessors is to obtain high performance. With a concrete understanding of how the decomposition, assignment, and orchestration of a parallel program are incorporated in the code that runs on the machine, we are ready to examine the key factors that limit parallel performance and how they are addressed in a wide range of problems. We will see how decisions made in different steps of the programming process affect the run-time characteristics presented to the architecture, as well as how the characteristics of the architecture influence programming decisions. Understanding programming techniques and these interdependencies is important not only for parallel software designers but also for architects. Besides understanding parallel programs as workloads for the systems we build, we learn to appreciate hardware/software trade-offs. In particular, we learn which aspects of programmability and performance the architecture can positively impact and which aspects are best left to software. The interdependencies of program and system are more fluid, more complex, and more important to performance in multiprocessors than in uniprocessors; hence, this understanding is critical to our goal of designing high-performance systems that reduce cost and programming effort. We carry it with us throughout the book, starting with concrete guidelines for workload-driven architectural evaluation in Chapter 4.

The space of performance issues and techniques in parallel software is very rich: different goals trade off with one another, and techniques that further one goal may cause us to revisit the techniques used to address another. This is what makes the creation of parallel software so interesting and challenging. As in uniprocessors, most performance issues can be addressed either by algorithmic and programming techniques in software or by architectural techniques or both. The focus of this chapter is on performance issues and software techniques. Architectural techniques, sometimes hinted at here, are the subject of the rest of the book.

Although several interacting performance issues must be considered, they are not dealt with all at once. The process of creating a high-performance program is one of successive refinement. As discussed in Chapter 2, the partitioning steps—decomposition and assignment—are often largely independent of the underlying architecture or programming model and concern themselves with major algorithmic issues that depend only on the inherent properties of the problem. In particular, these steps view the multiprocessor as simply a set of processors that communicate with one another. Their goal is to resolve the tension between balancing the workload across processes, reducing the interprocess communication inherent in the program, and