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

reducing the extra work needed to compute and manage the partitioning. We focus our attention first on addressing these partitioning issues.

Next, we open up the architecture and examine the new performance issues it raises for the orchestration and mapping steps. Opening up the architecture means recognizing two facts. The first fact is that a multiprocessor is not only a collection of processors but also a collection of memories, which an individual processor can view as an extended memory hierarchy. The management of data in these memory hierarchies can cause more data to be transferred across the network than the inherent communication mandated by the partitioning in the parallel program. The actual communication that occurs therefore depends both on the partitioning and on how the program's access patterns and locality of data reference interact with the organization and management of the extended memory hierarchy. The second fact is that the cost of communication as seen by the processor—and hence the contribution of communication to the execution time of the program—depends not only on the amount of communication but also on how it is structured to interact with the architecture. Section 3.2 discusses the relationship between communication, data locality, and the extended memory hierarchy. Then Section 3.3 examines the software techniques to address the major performance issues in orchestration and mapping: reducing the extra communication by exploiting data locality in the extended memory hierarchy and structuring communication to reduce its cost.

Of course, the architectural interactions and communication costs that we must deal with in orchestration sometimes cause us to go back and revise our partitioning methods, which is an important part of the refinement in parallel programming. Whereas interactions and trade-offs take place among all the performance issues we discuss, this chapter addresses each issue independently as far as possible and identifies trade-offs as they are encountered. Examples are drawn throughout from the four case study applications, and the impact of some individual programming techniques is illustrated through measurements on a cache-coherent machine with physically distributed memory, the Silicon Graphics Origin2000 (which is described in detail in Chapter 8). The equation solver kernel is also carried through the discussion, and performance techniques are applied to it as relevant; by the end of the discussion we will have created a high-performance parallel version of the solver.

As we examine the performance issues, we will develop simple analytical expressions for the speedup of a parallel program and illustrate how each performance issue affects the speedup equation. However, from an architectural perspective, a more concrete way of looking at performance is to examine the different components of execution time as seen by an individual processor in a machine—that is, how much time the processor spends executing instructions, accessing data in the extended memory hierarchy, and waiting for synchronization events to occur. In fact, these components of execution time can be mapped directly to the performance issues that software must address in the steps of creating a parallel program. Examining this view of performance helps us understand very concretely what a parallel execution looks like as a workload presented to the architecture, and the mapping helps us understand how programming techniques can alter this profile. These topics are discussed in Section 3.4.

Once we have studied the performance issues and techniques, we will be ready to understand how to create high-performance parallel versions of real applications—namely, the four case studies. Section 3.5 applies the parallelization process and performance techniques to each case study in turn. It illustrates how the techniques are employed together as well as the range of resulting execution characteristics that are presented to an architecture, reflected in varying profiles of execution time. We will also be ready to consider the implications of realistic applications for trade-offs between the two major lower-level programming models: a shared address space and explicit message passing. The trade-offs are in ease of programming and in performance and are discussed in Section 3.6. Let us begin with the algorithmic performance issues in the decomposition and assignment steps.

3.1 PARTITIONING FOR PERFORMANCE

For these steps, we can view the machine as simply a set of cooperating processors, largely ignoring its programming model and organization. All we need to know at this stage is that communication between processors is expensive. The three primary algorithmic issues are

- *balancing the workload* and reducing the time spent waiting at synchronization events
- *reducing communication*
- *reducing the extra work* done to determine and manage a good assignment

Unfortunately, even the three primary algorithmic goals are at odds with one another and must be traded off. A singular goal of minimizing communication would be satisfied by running the program on a single processor, as long as the necessary data fits in the local memory, but this would yield the ultimate load imbalance. On the other hand, near perfect load balance could be achieved—at a tremendous communication and task management penalty—by making each primitive operation in the program a task and assigning tasks randomly. And in many complex applications, load balance and communication could be improved by spending more time determining a good assignment, which results in extra work. The goal of decomposition and assignment is to achieve a good compromise between these conflicting demands as we see illustrated in the case studies and the equation solver kernel.

3.1.1 Load Balance and Synchronization Wait Time

In its simplest form, balancing the workload means ensuring that every processor does the same amount of work. It extends exposing enough concurrency (which we saw in Chapter 2 when discussing Amdahl's Law) with proper assignment and reduced serialization, and it gives the following simple limit on potential speedup:

$$\text{Speedup}_{problem}(p) \leq \frac{\text{Sequential Work}}{\max \text{ Work on Any Processor}}$$

Work in this context should be interpreted liberally because what matters is not only how many calculations are done but also the time spent doing them, which involves data accesses and communication as well.

In fact, load balancing is a little more complicated than simply equalizing work. Not only should different processors do the same amount of work, they should also be working at the same time. The extreme point would be if the work were evenly divided among processes but only one process were active at a time so there would be no speedup at all! The real goal of load balance is to minimize the time processes spend waiting at synchronization points, including an implicit one at the end of the program. This also involves minimizing the serialization of processes because of either mutual exclusion (waiting to enter critical sections) or dependences. The assignment step should ensure that low serialization is possible, and orchestration should ensure that it happens.

The process of balancing the workload and reducing synchronization wait time consists of four parts:

1. Identifying enough concurrency in decomposition and overcoming Amdahl's Law
2. Deciding how to manage the concurrency (statically or dynamically)
3. Determining the granularity at which to exploit the concurrency
4. Reducing serialization and synchronization cost

This section examines some techniques for each, using examples from the four case studies and other applications as well.

Identifying Enough Concurrency: Data and Function Parallelism

We saw in parallelizing the equation solver kernel that concurrency may be found by examining the loops of a program, by looking more deeply at the fundamental dependences, or by exploiting an understanding of its underlying problem to discover algorithms that afford more concurrency. Parallelizing loops often leads to similar (not necessarily identical) operation sequences or functions being performed on elements of a large data structure, as in the equation solver kernel. This is called *data parallelism* and is a more general form of the parallelism that inspired data parallel architectures discussed in Chapter 1. Computing forces on different particles in Barnes-Hut is another example.

In addition to data parallelism, applications often exhibit *function parallelism* as well: entirely different calculations can be performed concurrently on either the same or different data. Function parallelism is often referred to as control parallelism or task parallelism, though these are overloaded terms. For example, setting up an equation system for the solver in Ocean requires many different computations on ocean cross sections, each using a few cross-sectional grids. Analyzing dependences at the level of entire grids or arrays reveals that several of these computations are independent of one another and can be performed in parallel. Pipelining is another form of function parallelism in which different functions or stages of the pipeline are

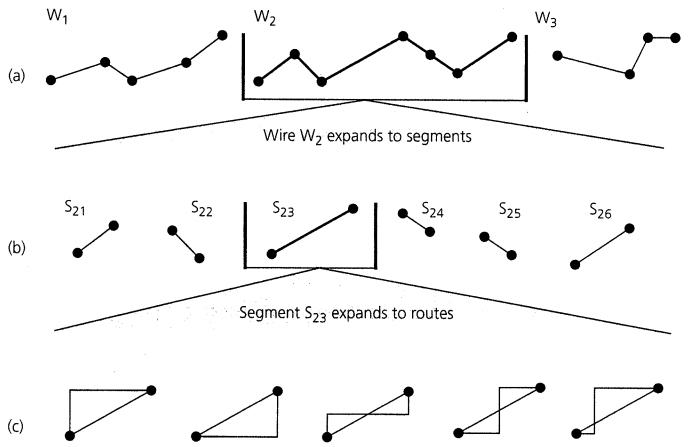


FIGURE 3.1 The three axes of parallelism in a VLSI wire-routing application: (a) wire parallelism; (b) segment parallelism; (c) route parallelism. The filled circles indicate the pins that are connected by wires.

performed concurrently on different data. For example, in encoding a sequence of video frames, each block of each frame passes through several stages: prefiltering, convolution from the time to the frequency domain, quantization, entropy coding, and so on. Pipeline parallelism is available across these stages (for example, a few processes could be assigned to each stage and operate concurrently), as is data parallelism between frames, among blocks in a frame, and within an operation on a block.

Function parallelism and data parallelism are often available together in an application and provide a hierarchy of levels of parallelism from which we must choose (e.g., function parallelism across grid computations and data parallelism within grid computations in Ocean, and the video encoding example). Orthogonal levels of data or function parallelism are found in many other applications as well; for example, applications that route wires in VLSI circuits exhibit parallelism across the wires to be routed, across the two-pin segments within a wire, and across the many routes evaluated for each segment (see Figure 3.1).

The degree of available function parallelism is usually modest and does not grow much with the size of the problem being solved. The degree of data parallelism, on the other hand, usually grows with data set size. Function parallelism is also usually more difficult to exploit in a load-balanced way, since different functions involve different amounts of work and have different scaling characteristics. Most parallel programs that run on large-scale machines are data parallel according to our loose definition of the term, and exploit function parallelism mainly to reduce the amount

of global synchronization required between data parallel computations (as illustrated in Ocean in Section 3.5.1).

By identifying the different types of concurrency available in an application, we often find much more concurrency than we need for load balancing. The next step in decomposition is to restrict the available concurrency by determining the granularity of tasks. However, the choice of task size also depends on how we expect to manage the concurrency, so let us discuss this next.

Determining How to Manage Concurrency: Static versus Dynamic Assignment

A key issue in exploiting concurrency is whether a good load balance can be obtained by a static or predetermined assignment (introduced in Chapter 2) or whether more dynamic means are required. A static assignment is typically an algorithmic mapping of tasks to processes, as in the simple equation solver kernel discussed in the previous chapter. Exactly which tasks (grid points or rows) are assigned to which processes may depend on the problem size, the number of processes, and other parameters, but once it is determined, the assignment does not change again at run time. Since the assignment is predetermined, static techniques do not incur much task management overhead at run time. However, to achieve good load balance, they require that the relative amounts of work in different tasks be adequately predictable or that enough tasks exist to ensure a balanced distribution by virtue of the statistics of large numbers. In addition to the program itself, it is also important that other environmental conditions—such as interference from other applications—not perturb the relationships among processors, thus limiting the robustness of static load balancing.

Dynamic partitioning techniques adapt to load imbalances at run time. They come in two forms. In *semistatic* techniques, the assignment for a phase of computation is determined algorithmically before that phase, but assignments are recomputed periodically to restore load balance based on profiles of the actual workload distribution gathered at run time. For example, we can profile (measure) the work that each task does in one phase and use that as an estimate of the work associated with it the next time that phase is executed. This repartitioning technique is used to assign stars to processes in Barnes-Hut (Section 3.5.2) by using profiles to recompute the assignment between time-steps of the galaxy's evolution. The galaxy evolves slowly, so the workload distribution among stars does not change much between successive time-steps. Figure 3.2(a) illustrates the advantage of semistatic partitioning over a static assignment of particles to processors, for a 512-K particle execution measured on the Origin2000, despite the cost of periodic repartitioning. It is clear that the performance difference grows with the number of processors used.

The second dynamic technique, *dynamic tasking*, is used to handle cases in which either the work distribution or the system environment is too unpredictable even to

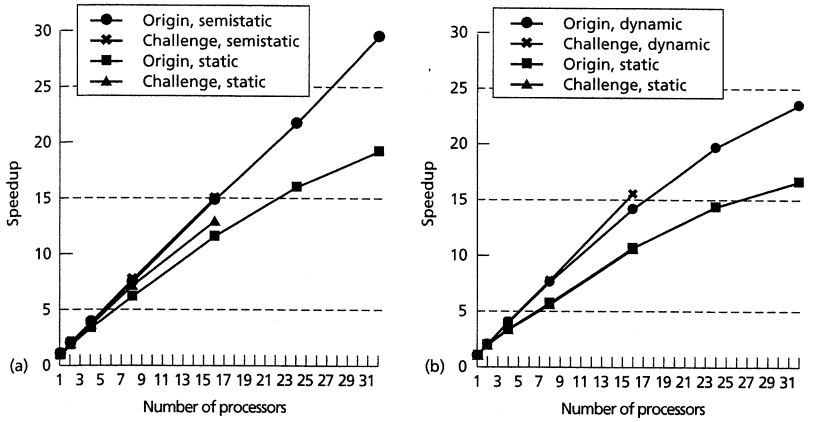


FIGURE 3.2 Illustration of the performance impact of dynamic partitioning for load balance. The graph in (a) shows the speedups of the Barnes-Hut application with and without semistatic partitioning, and the graph in (b) shows the speedups of Raytrace with and without dynamic tasking. Even in these applications that have a lot of parallelism, dynamic partitioning is important for improving load balance over static partitioning.

periodically recompute a load-balanced assignment.[1] For example, in Raytrace the work associated with each ray is impossible to predict. Even if the rendering is repeated from different viewpoints, the change in viewpoints may not be gradual. The dynamic tasking approach divides the computation into tasks and maintains a pool of available tasks (in Raytrace a task may be a ray or a set of rays). Each process repeatedly takes a task from the pool and executes it—possibly inserting new tasks into the pool—until no tasks are left. Of course, the management of the task pool must preserve the dependences among tasks—for example, by inserting a task only when it is ready for execution. Since dynamic tasking is widely used, let us look at some specific techniques to implement the task pool. Figure 3.2(b) illustrates the advantage of dynamic tasking over a static assignment of rays to processors in the

1. The applicability of static or semistatic assignment depends not only on the computational properties of the program but also on its interactions with the memory and communication systems and on the predictability of the execution environment. For example, differences in memory or communication stall time (due to cache misses, page faults, or contention) can cause imbalances observed at synchronization points even when the workload is computationally load balanced. Static assignment also may not be appropriate for time-shared or heterogeneous systems.

Raytrace application, for a data set consisting of number of balls arranged like a bunch of grapes, measured on the Origin2000.

A simple example of dynamic tasking in a shared address space is *self-scheduling* of a parallel loop. The loop counter is a shared variable accessed by all the processes that execute iterations of the loop. Processes obtain a loop iteration by incrementing the counter atomically; they repeatedly execute an iteration and access the counter again until no iterations remain. The task size can be increased by taking multiple iterations at a time, that is, adding a value larger than one to the shared loop counter. However, this can increase load imbalance. In *guided self-scheduling* (Aiken and Nikolau 1988), processes start by taking large chunks and taper down the chunk size as the loop progresses, hoping to reduce the number of accesses to the shared counter without compromising load balance.

More general dynamic task pools are usually implemented by a collection of queues into which tasks are inserted and from which tasks are removed and executed by processes. This may be a single centralized queue or a set of distributed queues, typically one per process, as shown in Figure 3.3. A *centralized queue* is simpler but has the disadvantage that every process accesses the same task queue, potentially increasing communication and causing processors to contend for queue access. Modifications to the queue (enqueuing or dequeuing tasks) must be mutually exclusive, further increasing contention and causing serialization. Unless tasks are large, and therefore queue accesses are few relative to computation, a centralized queue can quickly become a performance bottleneck as the number of processors increases.

With *distributed queues*, every process is initially assigned a set of tasks in its local queue. This initial assignment may be done intelligently to reduce interprocess communication, thus providing more control than self-scheduling and centralized queues. A process removes and executes tasks from its local queue as far as possible. If it creates tasks, it inserts them in its local queue. When no more tasks are in its local queue, it queries other processes' queues to obtain tasks from them, a mechanism known as *task stealing*. Because task stealing implies communication and can generate contention, several interesting issues arise in implementing stealing: for example, how to minimize stealing, whom to steal from, how many and which tasks to steal at a time, and so on. Stealing also introduces the important issue of *termination detection:* how do we decide when to stop searching for tasks to steal and assume that they're all done, given that tasks generate other tasks that are dynamically inserted in the queues? Simple heuristic solutions to this problem work well in practice, although a robust solution can be quite subtle and communication intensive (Dijkstra and Sholten 1968; Chandy and Misra 1988). Task queues are used both in a shared address space, where the queues are shared data structures that are manipulated using locks, and with explicit message passing, where the owners of queues service requests for them.

Although dynamic techniques generally provide good load balancing despite unpredictability or environmental conditions, they make the management of parallelism more expensive. Dynamic tasking techniques also compromise the explicit control over which tasks are executed by which processes, thus potentially increasing communication and compromising data locality. Static techniques are therefore usually preferable when they can provide good load balance for an application and environment.

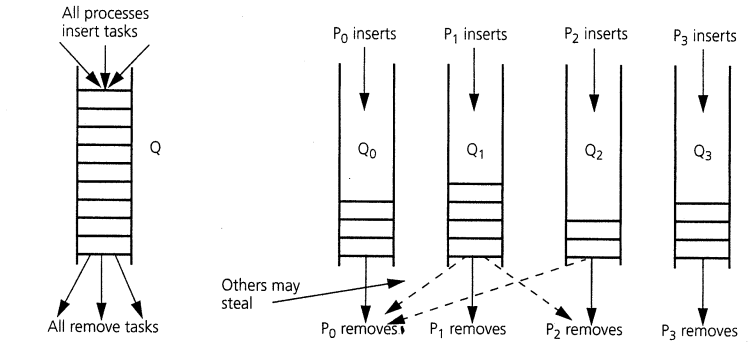


FIGURE 3.3 Implementing a dynamic task pool with a system of task queues

Determining the Granularity of Tasks

If no load imbalances occur due to dependences among tasks (for example, if all tasks are ready to be executed at the beginning of a phase of computation), then the maximum load imbalance possible with a task-queue strategy is equal to the granularity of the largest task. By *task granularity,* we mean the amount of work associated with a task, which is measured by the number of instructions or, more appropriately, the execution time. The general rule for choosing a granularity at which to actually exploit concurrency is that fine-grained or small tasks have the potential for better load balance (more tasks to divide among processes and hence more concurrency), but they lead to higher task management overhead, more contention, and more interprocessor communication than coarse-grained or large tasks. Let us see why, first in the context of dynamic task queuing where the definitions and trade-offs are clearer.

Task Granularity with Dynamic Task Queuing Here, a task is explicitly defined as an entry placed on a task queue, so task granularity is the work associated with such an entry. The larger task management (queue manipulation) overhead with small tasks is clear. At least with a centralized queue, the more frequent need for queue access generally leads to greater contention as well. Finally, breaking up a task into

two smaller tasks might cause the two tasks to be executed on different processors, thus increasing communication if the tasks access the same logically shared data.

Task Granularity with Static Assignment With static assignment, tasks are not explicit in the program, so it is less clear what should be called a task or a unit of concurrency. For example, in the equation solver, is a task a group of rows, a single row, or an individual element? We can define a task as the largest unit of work such that even if the assignment of tasks to processes is changed, the code that implements a task need not change. With static assignment, task size has a much smaller effect on task management overhead compared to dynamic task queuing since there are no queue accesses. Communication and contention are affected by the assignment of tasks to processors, not their size. The major impact of task size is usually on load imbalance and on exploiting data locality in processor caches.

Reducing Serialization

Finally, to reduce serialization at synchronization points, whether it is due to mutual exclusion or dependences among tasks, we must be careful about how we assign tasks as well as how we orchestrate synchronization and schedule tasks. For event synchronization, an example of excessive serialization is the use of more conservative synchronization than necessary, such as barriers instead of point-to-point or group synchronization. Even if point-to-point synchronization is used, it may preserve data dependences at a coarser grain than is required; for example, a process waits for another to produce a whole row of a matrix when the actual dependences are at the level of individual matrix elements. However, finer-grained synchronization is often more complex to program; it also implies the execution of more synchronization operations (say, one per word rather than one per larger data structure), the overhead of which may turn out to be more expensive than the savings in serialization. As usual, trade-offs abound.

For mutual exclusion, we can reduce serialization by using separate locks for separate data items and making the critical sections protected by locks smaller and less frequent if possible. Consider the former technique. In a database application, we may want to lock when we update certain fields of records that are assigned to different processes. The question is how to organize the locking. Should we use one lock per process, one per record, or one per field? The finer the granularity, the lower the contention, but the greater the space overhead and the less frequent the reuse of locks. An intermediate solution is to use a fixed number of locks and share them among records using a simple hashing function from records to locks. Another way to reduce serialization is to stagger the critical sections in time, that is, to arrange the computation so that multiple processes do not try to access the same lock at the same time.

Implementing task queues provides an interesting example of making critical sections smaller and less frequent. Suppose each process adds a task to a queue, then searches the queue for another task with a particular characteristic, and then removes this latter task from the queue. The task insertion and deletion may need to

be mutually exclusive—or may not if they are done at different ends of the queue—but the searching of the queue does not. Thus, instead of using a single critical section for the whole sequence of operations, we can break it up into two critical sections (insertion and deletion) and use code that is not mutually exclusive to search the list in between.

More generally, checking (reading) the state of a protected data structure usually does not have to be done with mutual exclusion; only modifying the data structure does. If the common case is to check but not to modify, as for the tasks we search through in the task queue, we can check without locking and, only if the check returns the appropriate condition, then lock and recheck within the critical section (to ensure the state hasn't changed) before modifying. In addition, instead of using a single lock for the entire queue, we can use a lock per queue element so that elements in different parts of the queue can be inserted or deleted in parallel (without serialization). As with event synchronization, the correct trade-offs in performance and programming ease depend on the costs and benefits of the choices on a system.

We can extend our simple limit on speedup to reflect both load imbalance and time spent waiting at synchronization points as follows, where *max* in the denominator is the maximum over all processes:

$$\text{Speedup}_{problem}(p) \leq \frac{\text{Sequential Work}}{\max(\text{Work} + \text{Synch Wait Time})}$$

In general, the different aspects of balancing the workload are the responsibility of software. An architecture cannot do very much about a program that does not have enough concurrency or is not load balanced. However, an architecture can help in some ways. First, it can provide efficient support for load-balancing techniques, such as task stealing, that are used widely by parallel software (applications, libraries, and operating systems). An access to a remote task queue for stealing is usually a probe or query, involving a small amount of data transfer and perhaps mutual exclusion. The more efficient the support for fine-grained communication and for low-overhead, mutually exclusive access to data, the smaller we can make our tasks and thus improve load balance. Second, the architecture can make it easy to name or access the logically shared data that a stolen task needs. Third, the architecture can provide efficient support for point-to-point synchronization, making it more attractive to use this form of synchronization instead of conservative barriers and hence allowing better load balance to be achieved.

3.1.2 Reducing Inherent Communication

Load balancing by itself is conceptually quite easy as long as the application affords enough concurrency: we can simply make tasks small and use dynamic tasking. Perhaps the most important performance goal to be traded off with load balance is reducing interprocessor communication. Decomposing a problem into multiple tasks usually means that communication will be required among tasks. If these tasks are assigned to different processes, we incur communication among processes and

hence processors. The focus in this section is on reducing communication that is *inherent* to the parallel program (i.e., one process produces data values that another needs) while still preserving load balance, thus retaining the view of the machine as a set of cooperating processors. However, in a real system communication occurs for other reasons, as Section 3.2 shows.

The impact of communication is best estimated not by the absolute amount of communication but by a quantity called the *communication-to-computation ratio*. This is defined as the amount of communication (in bytes, say) divided by the computation time (or because time is influenced by many factors, by the number of instructions executed). For example, a gigabyte of communication has a much greater impact on the execution time and communication bandwidth requirements of an application if the time required for the application to execute is 1 second than if it is 1 hour! The communication-to-computation ratio may be computed as a per-process number or accumulated over all processes.

The inherent communication-to-computation ratio is primarily controlled by the assignment of tasks to processes. To reduce communication, we should try to ensure that tasks accessing the same data or requiring frequent communication with one another are assigned to the same process. For example, in a database application, communication would be reduced if queries and updates that access the same database records are assigned to the same process.

One partitioning principle that has worked very well in practice for load balancing and inherent communication is *domain decomposition*. It was initially used in data parallel scientific computations such as Ocean but has since been found applicable to many other areas. If the data set on which the application operates can be viewed as a physical domain, then it is often the case that a point in the domain requires information either directly from only a small localized region around that point or from a longer range, with the requirements falling off with increasing distance from the point. We saw an example of the latter in Barnes-Hut. For the former, consider a video application in which algorithms for motion estimation in video encoding and decoding examine only the areas of a scene that are close to the current pixel; similarly, a point in the equation solver kernel needs to access only its four nearest-neighbor points directly. The goal of partitioning in these cases is to give every process a contiguous region of the domain, while of course retaining load balance, and to shape the domain so that most of the process's information requirements are satisfied within its assigned partition. As Figure 3.4 shows, in many such cases the communication requirements for a process grow proportionally to the size of a partition's boundary, whereas computation grows proportionally to the size of its entire partition. The communication-to-computation ratio is thus a surface-area-to-volume ratio in three dimensions and a perimeter-to-area ratio in two dimensions. It can be reduced by either increasing the data set size (n^2 in the figure) or reducing the number of processors (p).

Of course, the ideal shape for partitions in a domain decomposition is application dependent, depending primarily on the information requirements of and work associated with the points in the domain. For the equation solver kernel, in Chapter 2 we

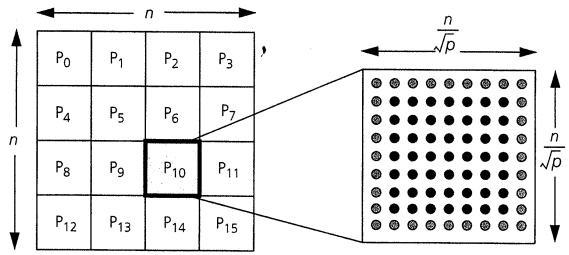


FIGURE 3.4 The perimeter-to-area relationship of communication to computation in a two-dimensional domain decomposition. The example shown is for an algorithm with localized, nearest-neighbor information exchange like the simple equation solver kernel. Every point on the grid needs information from its four nearest neighbors. Thus, the darker internal points in processor P_{10}'s partition do not need to communicate directly with any points outside the partition. Computation for processor P_{10} is thus proportional to the sum of all n^2/p points, whereas communication is proportional to the number of lighter boundary points, which is $4n/\sqrt{p}$.

chose to partition the grid into blocks of contiguous rows. Figure 3.5 shows that partitioning the grid into squarelike subgrids leads to a lower inherent communication-to-computation ratio. The impact becomes greater as the number of processors increases relative to the grid size. We shall therefore carry forward this partitioning into square subgrids (or simply "subgrids") as we continue to discuss performance. As a simple exercise, think about what the communication-to-computation ratio would be if we assigned rows to processes in an interleaved or cyclic fashion instead (row i assigned to process i mod *nprocs*).

How do we find a suitable domain decomposition that is load balanced and also keeps communication low? This can be accomplished statically or semistatically, depending on the nature and predictability of the computation:

- *Statically, by inspection*, as in the equation solver kernel and in Ocean. This requires predictability and usually leads to regularly shaped partitions, as in Figures 3.4 and 3.5.
- *Statically, by analysis*. The computation and communication characteristics may depend not only on the size of the input but also on the structure of the input presented to the program at run time, thus requiring an analysis of the input. However, the partitioning may need to be done only once after the input analysis—before the actual computation starts—so we still consider it static. Partitioning sparse matrix computations used in aerospace and automobile simulations is an example: the matrix structure is fixed but is highly irregular and requires sophisticated graph partitioning. Another example is Data Mining. Here, we may divide the database of transactions statically among processors, but a balanced assignment of itemsets to processes requires some analysis

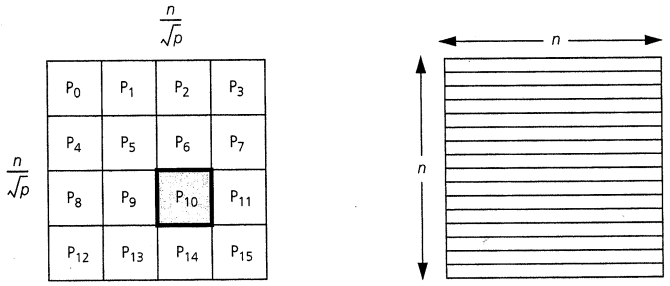


FIGURE 3.5 Choosing among domain decompositions for a simple nearest-neighbor computation on a regular two-dimensional grid. Since the work per grid point is uniform, equally sized partitions yield good load balance. But we still have choices. We might partition the elements of the grid into either strips of contiguous rows (right) or block-structured partitions that are as close to square as possible (left). The perimeter-to-area (and hence communication-to-computation) ratio in the block decomposition case is

$$\frac{4 \times n/\sqrt{p}}{n^2/p} \text{ or } \frac{4 \times \sqrt{p}}{n}$$

whereas that in strip decomposition is $\frac{2 \times n}{n^2/p}$ or $\frac{2 \times p}{n}$.

As p increases, block decomposition incurs less inherent communication for the same computation than strip decomposition.

since the work associated with different itemsets is not equal. A simple static assignment of itemsets and the database by inspection keeps communication low but does not provide load balance.

- *Semistatically, with periodic repartitioning.* This was discussed earlier for applications like Barnes-Hut whose characteristics change slowly with time. Domain decomposition is still important to reduce communication, as we see in the profiling-based Barnes-Hut case study in Section 3.5.2.
- *Statically or semistatically, with dynamic task stealing.* Even when the computation is highly unpredictable and dynamic task stealing must be used, domain decomposition may be useful in initially assigning tasks to processes. Raytrace is an example. Here there are two domains: the three-dimensional scene being rendered and the two-dimensional image plane. Since the natural tasks are rays shot through the image plane, it is much easier to manage domain decomposition of that plane than of the scene itself. We partition the image domain much like the grid in the equation solver kernel (Figure 3.4), with image pixels corresponding to grid points, and initially assign rays to the corresponding processes. This is useful because rays shot through adjacent pixels tend to access much of the same scene data. Processes then steal rays (pixels) or groups of rays dynamically for load balancing.

Of course, partitioning into a contiguous subdomain per processor is not always appropriate for high performance in all applications, as illustrated by the Gaussian elimination example in Exercise 3.9. Even Raytrace may benefit from dividing the image into more blocks than there are processors and assigning blocks to processors in an interleaved manner, trading off increased communication for better initial load balance. Different phases of the same application may also call for different partitioning. The range of techniques is very large, but common principles like domain decomposition can be found. For example, even when stealing tasks for load balancing in very dynamic applications, we can reduce communication by searching other queues in the same order every time or by preferentially stealing large tasks or several tasks at once to reduce the number of times we have to access nonlocal queues.

In addition to reducing communication volume, it is also important to keep communication (not just computation) balanced among processors. Since communication is expensive, imbalances in communication can translate directly to imbalances in execution time among processors. Overall, whether trade-offs in partitioning should be resolved in favor of load balance or communication volume depends on the cost of communication on a given system. Including communication as an explicit performance cost refines our basic speedup limit to

$$\text{Speedup}_{problem}(p) \leq \frac{\text{Sequential Work}}{\max(\text{Work} + \text{Synch Wait Time} + \text{Comm Cost})}$$

Compared to the previous expression, this expression separates communication from work, which now includes instructions executed plus local data access costs.

The amount of communication in parallel programs clearly has important implications for architecture. In fact, architects examine the needs of applications to determine what communication latencies and bandwidths are worth spending extra money for (see Exercise 3.14); for example, the bandwidth provided by a machine can usually be increased by throwing hardware (and hence money) at the problem, but this is only worthwhile if applications will exercise the increased bandwidth. As architects, we assume that the programs delivered to us are reasonable in their load balance and their communication demands, and we strive to make them perform better by providing the necessary support. Let us now examine the last of the algorithmic issues that we can resolve in partitioning itself without addressing the underlying architecture.

3.1.3 Reducing the Extra Work

The preceding discussion of domain decomposition suggests that when a computation is irregular, computing a good assignment that both provides load balance and reduces communication can be quite expensive. This extra work is not required in a sequential execution and is an overhead of parallelism. Consider the sparse matrix example that was discussed previously to illustrate static partitioning by analysis. The sparse matrix can be represented as a graph, such that each node represents a row or column of the matrix and an edge exists between two nodes i and j if the matrix entry (i,j) is nonzero. The goal in partitioning is to assign each process a set

of nodes such that the computation is load balanced and the number of edges that cross partition boundaries is minimized. Many clever partitioning techniques have been developed, but the ones that result in a better balance between load balance and communication require more time to partition the graph. We see this illustrated in the Barnes-Hut case study later in this chapter.

In addition to partitioning, another common source of extra work is redundant computation: multiple processes computing data values redundantly rather than having one process compute them and communicate them to the others, which may be a favorable trade-off when the cost of communication is high. Examples include all processes computing their own copy of the same shading table in computer graphics applications or of trigonometric tables in scientific computations. If the redundant computation can be performed while the processor is otherwise idle due to load imbalance, its cost can be hidden.

Finally, many aspects of orchestrating parallel programs involve extra work as well, such as creating processes, managing dynamic tasking, distributing code and data throughout the machine, executing synchronization operations and parallelism control instructions, structuring communication appropriately for a machine, and packing and unpacking data to and from communication messages. For example, the high cost of creating processes is what causes us to create them once up front and have them execute tasks until the program terminates, rather than creating and terminating processes as parallel sections of code are encountered and exited by a single main thread of computation (a *fork-join* approach, which is sometimes used with lightweight threads instead of processes). For example, in the Data Mining case study (Section 3.5.4), substantial extra work done to transform the database pays off in reducing communication, synchronization, and expensive input/output activity.

The trade-offs between extra work, load balance, and communication must be considered carefully when making partitioning decisions. The architecture can help reduce the need for extra work by making communication and task management more efficient. Based only on these algorithmic partitioning issues, the speedup limit can now be refined to

$$\text{Speedup}_{problem}(p) \leq \frac{\text{Sequential Work}}{\max(\text{Work} + \text{Synch Wait Time} + \text{Comm Cost} + \text{Extra Work})} \quad (3.1)$$

3.1.4 Summary

The analysis of parallel algorithm performance requires a characterization of a multiprocessor and a characterization of the parallel algorithm. Historically, the analysis of parallel algorithms has focused on algorithmic aspects like partitioning and mapping to network topologies and has not taken other architectural interactions into account. In fact, the most common model used to characterize a multiprocessor for algorithm analysis has been the Parallel Random Access Memory (PRAM) model (Fortune and Wyllie 1978). In its most basic form, the PRAM model

assumes that data access is free, regardless of whether it is local or involves communication. That is, communication cost is zero in the speedup expression of Equation 3.1, and work is treated simply as instructions executed:

$$\text{Speedup-PRAM}_{problem}(p) \leq \frac{\text{Sequential Instructions}}{\max(\text{Instr} + \text{Synch Wait Time} + \text{Extra Instr})} \quad (3.2)$$

A natural way to think of a PRAM model is as a shared address space machine in which all data access is free. The performance factors that matter in parallel algorithm analysis using this model are load balance (including serialization) and extra work. The goal of algorithm development for PRAMs is to expose enough concurrency so the workload may be well balanced without needing too much extra work.

While the PRAM model is useful in discovering the concurrency available in an algorithm, which is the first step in parallelization, it is clearly unrealistic for modeling performance on real parallel systems. This is because communication, which it ignores, can easily dominate the cost of a parallel execution in modern systems, and imbalances in communication cost can dominate imbalances in instructions executed. In fact, analyzing algorithms while ignoring communication can easily lead to a poor choice of decomposition and assignment, to say nothing of orchestration. More recent models have been developed to include communication costs as explicit parameters that algorithm designers can use (Valiant 1990; Culler et al. 1993). We return to this issue after we have a better understanding of communication costs.

The treatment of communication costs in this section is simplified in two respects relative to real systems. First, communication inherent to the parallel program and its partitioning is not the only form of communication that is important: substantial noninherent or *artifactual* communication may occur that is caused by interactions of the program with the architecture on which it runs. Thus, we have not yet modeled the amount of communication generated by a parallel program satisfactorily. Second, the communication cost term in Equation 3.1 is determined not only by the amount of communication caused, whether inherent or artifactual, but also by the structure of the communication in the program and how it interacts with the costs of the basic communication operations in the machine. Both artifactual communication and communication structure are important performance issues that are usually addressed in the orchestration step since they are architecture dependent. To understand them we first need a deeper understanding of some critical interactions of parallel architectures with parallel software.

3.2 DATA ACCESS AND COMMUNICATION IN A MULTIMEMORY SYSTEM

In our discussion of partitioning, we have viewed a multiprocessor as a collection of cooperating processors. However, multiprocessor systems are also multimemory, multicache systems, and the role of these components is essential to performance. The role is essential regardless of programming model, though the latter may influence the nature of the specific performance trade-offs. Our discussion turns to the

3.5 THE PARALLEL APPLICATION CASE STUDIES: AN IN-DEPTH LOOK

Having discussed the major performance issues for parallel programs in a general context and having applied them to the simple equation solver kernel, we are ready to examine how to achieve good parallel performance on more realistic applications on real multiprocessors. In particular, we now return to the four application case studies that motivated us to study parallel software in Chapter 2, apply the four steps of the parallelization process to each case study, and at each step address the major performance issues that arise there. In the process, we can understand and respond to the trade-offs among the different performance issues as well as between performance and ease of programming. Examining the components of execution time on a real machine will also help us see the types of workload characteristics that different applications present to a parallel architecture. Understanding the relationship between parallel applications, software techniques, and workload characteristics will be very important as we proceed through the rest of the book.

Parallel applications come in various shapes and sizes with very different characteristics and trade-offs among performance issues. Our four case studies provide an interesting though necessarily restricted cross section through the application space. In examining how to parallelize and, particularly, orchestrate the applications for good performance, we shall focus for concreteness on a specific architectural style: a cache-coherent shared address space multiprocessor with main memory physically distributed among the processing nodes.

The discussion of each application is divided into four subsections. The first describes in more detail the sequential algorithms and the major data structures used. The second describes the partitioning of the application (i.e., the decomposition of the computation and its assignment to processes), addressing the algorithmic performance issues of load balance, communication volume, and the overhead of computing the assignment. The third subsection is devoted to orchestration: it describes the spatial and temporal locality in the program as well as the synchronization used and the amount of work done between synchronization points. The fourth subsection discusses mapping to a network topology. Finally, for illustration we present the components of execution time as obtained for a real execution (using a particular problem size) on a specific machine of the chosen style: a 32-processor Silicon Graphics Origin2000. The busy-useful and busy-overhead components cannot be separated from each other in measurements on this machine, and neither can the data-local and data-remote components, so execution time is divided into three components: busy, data wait, and synchronization. While the level of detail at which we treat the case studies may appear high in some places, these details will be important in explaining the experimental results we shall obtain in later chapters using these applications.

3.5.1 Ocean

Ocean, which simulates currents in an ocean basin, resembles many important applications in computational fluid dynamics. Several of its properties are also representative of a wide range of applications, both scientific and commercial, that stream through large data structures and perform little computation at each data point. At each horizontal cross section through the ocean basin, several different variables are modeled, including the current, temperature, pressure, and friction. Each variable is discretized and represented by a regular, uniform two-dimensional grid of size $(n + 2)$-by-$(n + 2)$ points ($n + 2$ is used instead of n so that the number of internal, nonborder points that are actually updated in the equation solver is n-by-n). In all, about 25 different grid data structures are used by the application.

The Sequential Algorithm

After the currents at each cross section are initialized, the outermost loop of the application proceeds over a large, user-defined number of time-steps. Every time-step first sets up and then solves partial differential equations on the grids. A time-step consists of 33 different grid computations, each involving one or a small number of grids (variables). Typical grid computations include adding together scalar multiples of a few grids and storing the result in another grid (e.g., $A = \alpha_1 B + \alpha_2 C - \alpha_3 D$), performing a single nearest-neighbor averaging sweep over a grid and storing the result in another grid, and solving a system of partial differential equations on a grid using an iterative method.

The iterative equation solver used is the multigrid method. This is a complex but efficient variant of the simple equation solver kernel we have discussed so far. In the simple solver, each sweep traverses the entire n-by-n grid (ignoring the border columns and rows). A multigrid solver, on the other hand, performs sweeps over a hierarchy of grids. The original n-by-n grid is the finest-resolution grid in the hierarchy; the grid at each coarser level removes every alternate grid point in each dimension, resulting in grids of size $n/2$-by-$n/2$, $n/4$-by-$n/4$, and so on. The first sweep of the solver traverses the finest grid, and successive sweeps are performed on coarser or finer grids depending on the error computed in the previous sweep, terminating when the system converges within a user-defined tolerance on the finest grid. To keep the computation deterministic and make it more efficient, a red-black ordering is used (see Section 2.3.2).

Decomposition and Assignment

Ocean affords concurrency at two levels within a time-step: across grid computations (function parallelism) and within a single grid computation (data parallelism). Little concurrency is available across successive time-steps. Concurrency across grid computations can be discovered by writing down which grids each computation reads and writes and analyzing the data dependences among them at this level. The

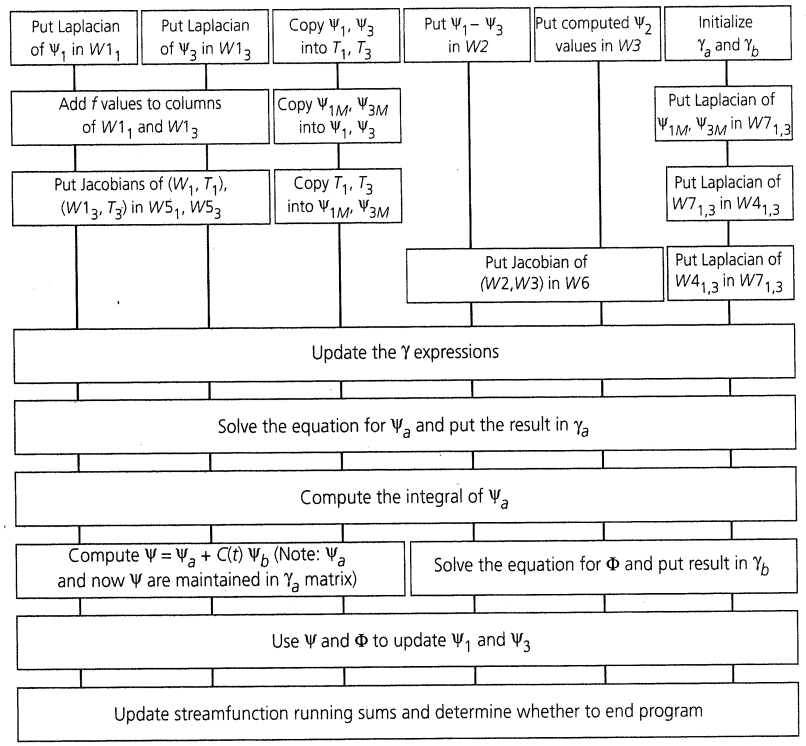


FIGURE 3.14 Ocean: The phases in a time-step and the dependences among grid computations. Each box is a grid computation (or pair of similar computations). Computations connected by vertical lines are dependent while others, such as those in the same row, are independent. The parallel program treats each horizontal row as a phase and synchronizes between phases.

resulting dependence structure and concurrency are depicted in Figure 3.14. Clearly, there is not enough concurrency across grid computations (i.e., not enough vertical sections) to occupy more than a few processors. We must therefore exploit the data parallelism within a grid computation as well, and we need to decide what combination of function and data parallelism is best.

In this case study, we choose to have all processes collaborate on each grid computation rather than to divide the processes among the available concurrent grid computations and use both levels of parallelism. Combined data and function parallelism would increase the size of each process's partition of a grid and hence reduce the communication-to-computation ratio. However, the work associated with different grid computations is quite varied and also depends on problem size in different ways, which complicates load balancing. Second, since several different computations in a time-step access the same grid, for communication and data locality reasons we would not like the same grid to be partitioned in different ways among processes in different computations. Third, all the grid computations are fully data parallel, and all grid points in a given computation do roughly the same amount of work, so we can statically assign grid points to processes. Nonetheless, knowing which grid computations are independent is useful because it allows processes to avoid synchronizing between them (see Figure 3.14).

The issues regarding inherent communication are very similar to those in the simple equation solver, so we use a block-structured (squarelike) domain decomposition of each grid. There is one complication—a trade-off between data locality and load balance related to the points at the border of the grid in some grid computations. The internal n-by-n points do similar work and are divided equally among all processes. Complete load balancing demands that border points, which often do less work, also be divided equally among processors. However, communication and data locality suggest that border points should be assigned to the processes that own the nearest internal points, which assign no border elements to several of the processes. We follow the latter strategy, incurring a slight load imbalance.

Finally, let us examine the multigrid equation solver. The grids at all levels of the multigrid hierarchy are partitioned in the same block-structured domain decomposition. However, the number of grid points per process decreases as we go to coarser levels of the hierarchy, so at the highest levels, some processes may become idle. Fortunately, relatively little (if any) time is spent at these load-imbalanced levels. The ratio of communication to computation also increases at higher levels since there are fewer points per process. This illustrates the importance of measuring speedups relative to the best sequential algorithm (here multigrid): a classical, nonhierarchical parallel iterative solver on the original (finest) grid would likely yield better *self-relative* speedups (relative to a single processor performing the same computation) than the parallel multigrid solver, but the multigrid solver is far more efficient sequentially and overall. In general, less efficient sequential algorithms often yield better self-relative speedups, but these are not useful measures for an end user.

Orchestration

Here we are mostly concerned with artifactual communication, data locality, and synchronization. Let us consider issues related to spatial locality first, then temporal locality, and finally synchronization.

Spatial Locality Within a grid computation, the issues related to spatial locality are similar to those of the simple equation solver kernel in Section 3.3.1. A four-dimensional array data structure is therefore used to represent each grid. This

results in very good spatial locality, particularly on local data. Accesses to nonlocal data (the elements at the boundaries of neighboring partitions) yield good spatial locality along row-oriented partition boundaries and poor locality (hence fragmentation or waste in communication) along column-oriented boundaries. One major difference between the simple solver and the complete Ocean application is that Ocean involves 33 different grid computations in every time-step, each involving one or more out of 25 different grids, so we experience many cache conflict misses across grids. These conflict misses are reduced by ensuring that the allocated dimensions of the arrays are not powers of two (even if the program uses power-of-two grids), but it is difficult to lay out different grids relative to one another to minimize conflict misses. A second difference has to do with the multigrid solver. Since a process's partition has fewer grid points at higher levels of the grid hierarchy, spatial locality is reduced and it is more difficult to distribute data appropriately among main memories at page granularity, despite the use of four-dimensional arrays.

Working Sets and Temporal Locality Ocean has a complicated working set hierarchy, with six working sets. The first two are due to the use of near-neighbor computations within a grid and are similar to those for the simple equation solver kernel. The first working set is captured when the cache is large enough to hold a few grid points so that a point that is accessed as the right neighbor for the previous point is reused to compute itself and to serve as the left neighbor for the next point. The second working set comprises three subrows of a process's partition. When the process returns from one subrow to the beginning of the next in a near-neighbor computation, it can reuse the elements of the previous subrow.

The rest of the working sets are not well defined as single working sets and do not produce sharp knees in the working set curve. The third working set constitutes a process's entire partition of a grid used in the multigrid solver. This could be the partition at any level of the multigrid hierarchy at which the process tends to iterate, so it is not really a single working set. The fourth working set consists of the sum of a process's subgrids at several successive levels of the grid hierarchy within which it tends to iterate (in the extreme, this becomes all levels of the grid hierarchy). The fifth working set allows reuse on a grid across grid computations or even phases; thus, it is large enough to hold a process's partition of several grids. The last working set holds all the data that a process is assigned in every grid so that all the data can be reused across time-steps.

The working sets that are most important to performance are the first three or four, depending on how the multigrid solver behaves. The largest among these grows linearly with the size of the data set per process. This growth rate is common in applications that repeatedly stream through their data sets, so with large data sets, some important working sets do not fit in the local caches. Fortunately, large data sets in these streaming applications make it easy to distribute data in memory at page granularity, so the working sets for a process consist mostly of local rather than communicated data. The little reuse that nonlocal data affords is captured by the first two working sets.

Synchronization Ocean uses two types of synchronization. First, global barriers are used to synchronize all processes between computational phases (see Figure 3.14) as well as between sweeps of the multigrid equation solver. Between several of the phases, we could replace the barriers with finer-grained point-to-point synchronization at the level of grid points to obtain some overlap across phases; however, in this case the overlap is likely to be too small to justify the programming complexity and the overhead of many more synchronization operations. The second form of synchronization is the use of locks to provide mutual exclusion for global reductions, for example, to determine convergence in the solver. The work between synchronization points is large, typically proportional to the size of a process's partition of a grid.

Mapping

Given the near-neighbor communication pattern, we would like to map processes to processors such that processes whose partitions are adjacent to each other in the grid run on processors that are near each other in the network topology. Our subgrid partitioning of two-dimensional grids clearly maps very well to a two-dimensional mesh network. However, as in all our programs, mapping of processes to processors is not enforced by the program but is left to the system.

Summary

Ocean is a good representative of many applications that stream through regular arrays. The computation-to-communication ratio is proportional to $n/\sqrt{p}$ for a problem with n-by-n grids and p processors, load balance is good except when n is not large relative to p, and the parallel efficiency for a given number of processors increases with the grid size. Since a processor streams through its portion of the grid in each grid computation, since only a few instructions are executed per access to grid data during each sweep, and since significant potential exists for conflict misses across grids, data distribution in main memory can be very important on machines with physically distributed memory.

Figure 3.15 shows the breakdown of execution time into busy, waiting at synchronization points, and waiting for data accesses to complete for a particular execution of Ocean with 1,030 × 1,030 grids using 2D and 4D arrays on a 32-processor SGI Origin2000 machine. This machine has very large per-processor second-level caches (4 MB), so with four-dimensional array representations each processor's partition tends to fit comfortably in its cache. The problem size is large enough relative to the number of processors that the inherent communication-to-computation ratio is quite low. The major bottleneck is the time spent waiting at barriers. Smaller problems would stress communication more, whereas larger problems and proper data distribution would put more stress on the local memory system. With two-dimensional arrays, the story is clearly different. Conflict misses are frequent, and with data being difficult to distribute appropriately in main memory many of these misses are not satisfied locally, leading to long latencies, contention, and high data wait time.

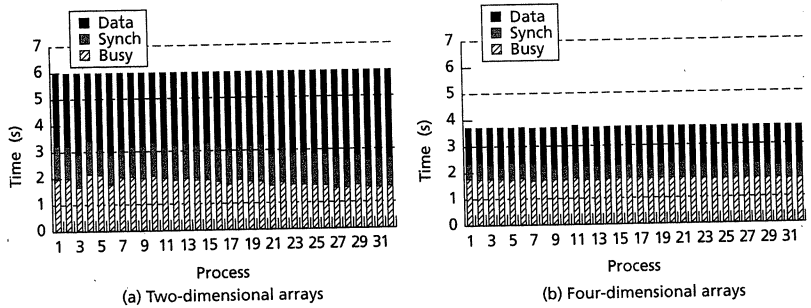


FIGURE 3.15 Execution time breakdown for Ocean on a 32-processor Origin2000. The size of each grid is 1,026 × 1,026, and the convergence tolerance is 10^{-3}. The use of four-dimensional arrays to represent the two-dimensional grids in (b) clearly reduces the time spent stalled on the memory system (including communication). This data wait time is very small because a processor's partition of the grids it uses at a given time fit very comfortably in the large 4-MB second-level caches in this machine. With smaller caches or much bigger grids, the time spent stalled waiting for (local) data would have been much larger. [8]

3.5.2 Barnes-Hut

The galaxy simulation has far more irregular and dynamically changing behavior than Ocean. Recall that it solves an n-body problem, in which the major computational challenge is to compute the influences that n bodies in a system exert on one another. The algorithm it uses for computing forces on the stars, the Barnes-Hut method, is an efficient hierarchical method for solving the n-body problem in $O(n \log n)$ time.

The Sequential Algorithm

The galaxy simulation proceeds over hundreds of time-steps, each step computing the net force on every body and thereby updating that body's position and other attributes. Recall the insight that the force calculation in the Barnes-Hut method is based on: if the magnitude of interaction between bodies falls off rapidly with distance (as it does in gravitation), so the effect of a large group of bodies may be approximated by a single equivalent body if that group of bodies is far enough away from the point at which the effect is being evaluated. The hierarchical application of this insight implies that the farther away the bodies, the larger the group that can be approximated by a single body.

8. In this and subsequent execution time breakdowns, there is no artificial final barrier to cause all processes to wait until the last is finished, as in Figure 3.12.

To facilitate a hierarchical approach, the Barnes-Hut algorithm represents the three-dimensional space containing the galaxies as a tree, as follows. The root of the tree represents a space cell containing all bodies in the system. The tree is built by adding bodies into the initially empty root cell and subdividing a cell into its eight equally sized children as soon as it contains more than a fixed number of bodies (here ten). The result is an oct-tree whose internal nodes are space cells and whose leaves contain the individual bodies.[9] Empty cells resulting from a cell subdivision are ignored. The tree (and the Barnes-Hut algorithm) is therefore adaptive in that it extends to more levels in regions that have high body densities. While we use a three-dimensional problem, Figure 3.16 shows a small two-dimensional example domain and the corresponding quadtree for simplicity. The positions of the bodies change across time-steps, so the tree is rebuilt every time-step. This results in the overall computational structure shown in Figure 3.17, with most of the time being spent in the force calculation phase.

The tree is traversed once per body to compute the net force acting on that body. The force calculation algorithm for a body starts at the root of the tree and conducts the following test recursively for every cell it visits. If the center of mass of the cell is far enough away from the body, the entire subtree under that cell is approximated by a single body at the center of mass of the cell, and the force this center of mass exerts on the body is computed. The rest of that subtree is not traversed. If, however, the center of mass is not far enough away, the cell must be "opened" and each of its subcells visited. A cell is determined to be far enough away if the following condition is satisfied:

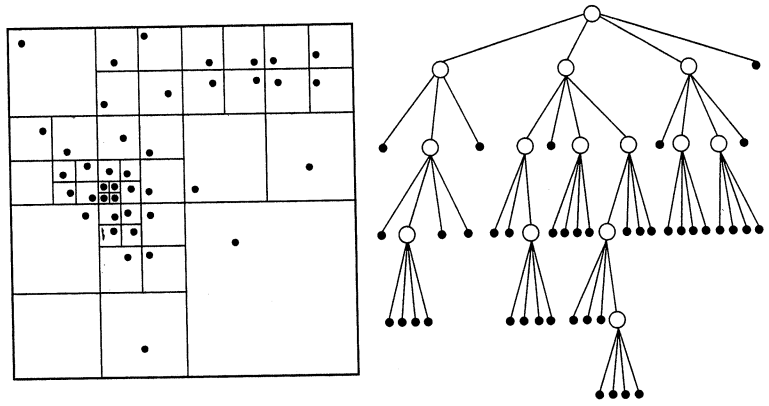
$$\frac{l}{d} < \theta \tag{3.6}$$

where l is the length of a side of the cell, d is the distance of the body from the center of mass of the cell, and θ is a user-defined accuracy parameter (θ is usually between 0.5 and 1.2). In this way, a body traverses deeper into those parts of the tree representing space that is physically close to it and groups distant bodies at a hierarchy of length scales. Since the expected depth of the tree is $O(\log n)$ and the number of bodies for which the tree is traversed is n, the expected complexity of the algorithm is $O(n \log n)$. Actually it is

$$O\left(\frac{1}{\theta^2} \times n \log n\right)$$

since θ determines the number of tree cells touched at each level in a traversal (smaller θ implies greater accuracy and more tree cells touched). Bodies in denser parts of the space traverse deeper down the tree to compute the forces on themselves, so the work associated with bodies is not uniform.

9. An oct-tree is a tree in which every node has a maximum of eight children. In two dimensions, a quadtree would be used, in which the maximum number of children is four.



(a) The spatial domain (b) Quadtree representation

FIGURE 3.16 Barnes-Hut: A two-dimensional particle distribution and the corresponding quadtree

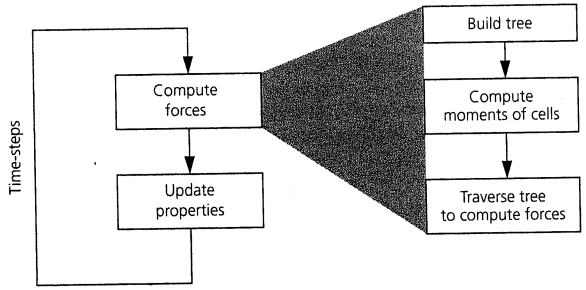


FIGURE 3.17 Flow of computation in the Barnes-Hut application. The force computation phase of an *n*-body problem expands into three phases (shown on the right) in the Barnes-Hut method.

Conceptually, the main data structure in the application is the Barnes-Hut tree. The tree is implemented in both the sequential and parallel programs with two arrays: an array of bodies and an array of tree cells. Each body and cell is represented as a structure or record. The fields for a body include its three-dimensional position, velocity, acceleration, and mass. A cell structure also has pointers to its children in the tree, and a three-dimensional center of mass. There is also a separate array of

pointers to bodies and one of pointers to cells. Every process owns a contiguous chunk of pointers in these arrays, not necessarily of equal size, which in every time-step are set to point to the bodies and cells that are assigned to it in that time-step. Since the structure and partitioning of the tree changes across time-steps as the galaxy evolves, the actual bodies and cells assigned to a process are not contiguous in the body and cell arrays.

Decomposition and Assignment

Each of the phases within a time-step is executed in parallel, with global barrier synchronization between phases. The natural unit of decomposition (task) in all phases is a body, except in computing the cell centers of mass, where it is a cell.

Unlike Ocean, which has a regular and predictable structure of both computation and communication, the Barnes-Hut application presents many challenges for effective assignment. First, the nonuniformity of the galaxy implies that both the amount of work per body and the communication patterns among bodies are nonuniform, so a good assignment cannot be discovered by inspection. Second, the distribution of bodies changes across time-steps, which means that static assignment is not likely to work well. Third, since the information needs in force calculation fall off with distance equally in all directions, reducing interprocess communication demands that partitions be spatially contiguous and not biased in size toward any one direction. Fourth, the different phases in a time-step have different distributions of work among the bodies/cells, and hence different preferred partitions. For example, the work in the update phase is uniform across all bodies, whereas that in the force calculation phase clearly is not. Another challenge for good performance is that the communication needed among processes is naturally fine grained and irregular.

We focus our partitioning efforts on the force calculation phase since it is by far the most time-consuming. The partitioning is not modified for other phases in accordance with their needs since the cost of doing so, both in repartitioning and in loss of locality, outweighs the potential benefits and since similar partitions are likely to work well for tree building and moment calculation phases (although not for the update phase).

We can use profiling-based semistatic partitioning in this application, taking advantage of the fact that although the spatial distribution of bodies at the end of the simulation may be radically different from that at the beginning, it evolves slowly with time and changes little between two successive time-steps. As we perform the force calculation phase in a time-step, we record the work done by every particle in that time-step (i.e., we count the number of interactions it computes with other bodies or cells). We then use this work count as a measure of the work associated with that particle in the next time-step. Work counting is cheap since it only involves incrementing a local counter when an (expensive) interaction is performed. Now we need to combine this load-balancing method with assignment techniques that also achieve the communication goal: keeping partitions contiguous in space and not biased in size toward any one direction. We briefly discuss two techniques:

the first because it is applicable to many irregular problems and the second because it is better suited to this application and is what our program uses.

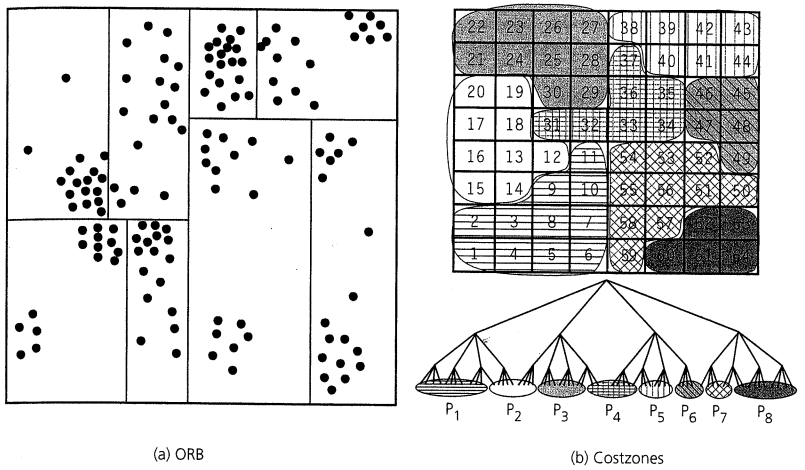
The first technique, called orthogonal recursive bisection (ORB), preserves physical locality by partitioning the domain space directly. The space is recursively subdivided into two rectangular subspaces with equal work, using the preceding load-balancing measure, until one subspace per process remains (see Figure 3.18[a]). Initially, all processes are associated with the entire domain space. Every time a space is divided, half the processes associated with it are assigned to each of the subspaces that result. The Cartesian direction in which division takes place is usually alternated with successive divisions, and a parallel median finder is used to determine where to split the current subspace. A separate binary tree of depth $\log p$ is used to keep track of the divisions and to implement ORB. (Details of using ORB for this application can be found in [Salmon 1990].)

The second technique, called costzones, recognizes that the Barnes-Hut algorithm already has a representation of the spatial distribution of bodies encoded in its tree data structure. Thus, we can partition this existing data structure itself and thereby achieve the goal of partitioning space (see Figure 3.18[b]). Every internal cell stores the total cost associated with all the bodies it contains. The total work or cost in the system is divided among processes so that every process has a contiguous, equal range or zone of work (for example, a total work of 1,000 units would be split among 10 processes so that zone 1–100 units is assigned to the first process, zone 101–200 to the second, and so on). Which costzone a body in the tree belongs to can be determined by the total cost of an in-order traversal of the tree up to that body. Processes traverse the tree in parallel, picking up the bodies that belong in their costzone. (Details can be found in [Singh et al. 1995].) The costzones method is much easier to implement than ORB. While the two result in partitions with similar load balance and inherent communication properties, the costzones method yields better overall performance in a shared address space. This is mostly because the time spent in the partitioning phase itself (i.e., computing the partitions) is much smaller, which illustrates the impact of extra work.

Orchestration

Orchestration issues in Barnes-Hut reveal many differences from Ocean, illustrating that even applications in scientific computing can have widely different behavioral characteristics of architectural interest.

Spatial Locality While the shared address space makes it easy for a process to access the parts of the shared tree that it needs in all the computational phases, distributing data to keep a process's assigned bodies and cells in its local main memory is not as easy as in Ocean. First, data would have to be redistributed dynamically as assignments change across time-steps, which can be expensive. Second, the logical granularity of data (a particle/cell) is much smaller than the physical granularity of allocation in memory (a page), and the fact that bodies/cells assigned to the same



(a) ORB (b) Costzones

FIGURE 3.18 Partitioning schemes for Barnes-Hut: ORB and costzones. ORB partitions space directly by recursive bisection, while costzones partition the tree. (b) shows both the partitioning of the tree and how the resulting space is partitioned by costzones. ORB leads to more regular (rectangular) partitions than costzones, but their communication and load balance properties are quite similar.

process are contiguous in physical space does not mean that they are spatially contiguous in the body/cell arrays. Fixing these problems requires overhauling the data structures that store bodies and cells: using separate arrays or lists per process that are modified across time-steps as assignments change, and hence different data structures than those used in the sequential program. Fortunately, there is enough temporal locality in the application that data distribution is not so important in a shared address space (again unlike Ocean). In addition, the vast majority of the cache misses are to bodies and cells that are assigned to other processors anyway, so data distribution itself wouldn't help make the misses local. We therefore simply distribute pages of shared data in a round-robin interleaved manner among nodes, without attention to which node gets which pages.

While in Ocean large cache blocks improve local access performance, limited only by partition size, here multiword cache blocks help exploit spatial locality only to the extent that reading a particle's displacement or moment data involves reading several double-precision words of data. Very large transfer granularities might cause more fragmentation than useful prefetching to occur for the same reason that data distribution at page granularity is difficult: unlike in Ocean, locality of bodies/cells in the arrays does not match that in physical space (on which assignment is based),

so fetching data from more than one body/cell in the array upon a miss may be harmful rather than beneficial. Spatial locality depends on the size of a body or cell structure and does not improve much with the number of bodies.

Working Sets and Temporal Locality The first working set in this program contains the data used to compute forces between a single body-body or body-cell pair. The interaction with the next body or cell in the traversal will reuse this data. The second working set is the most important to performance. It consists of the data encountered in the entire tree traversal to compute the force on a single body. Because of the way partitioning is done, the next body on which forces are calculated will be close to this body in space, so the tree traversal to compute the forces on that body will reuse most of this data. As we go from body to body, the composition of this working set changes slowly, but the amount of reuse is tremendous, and the resulting working set is small even though overall a process accesses a very large amount of data in irregular ways. Much of the data in this working set is from other processes' partitions, and most of this data is allocated nonlocally. Thus, it is the temporal locality exploited on shared data (both local and nonlocal) that is critical to the performance of the application, unlike in Ocean where it is data distribution.

By the same reasoning that the complexity of the algorithm is

$$O\left(\frac{1}{\theta^2} \times n \log n\right)$$

the expected size of this working set is proportional to

$$O\left(\frac{1}{\theta^2} \times \log n\right)$$

even though the overall memory requirement of the application is close to linear in n: each particle accesses about this much data from the tree to compute the force on it. The constant of proportionality is small, being the amount of data accessed from each body or cell visited during force computation. Since this working set grows slowly and fits comfortably in modern second-level caches, we do not need to replicate data in main memory. In Ocean, some important working sets grow linearly with the data set size, and we do not always expect them to fit in the cache; however, proper data distribution is easy and keeps most cache misses local, so even in Ocean we do not need replication in main memory.

Synchronization Barriers are used to maintain dependences among bodies and cells across some of the computational phases, such as between building the tree and using it to compute forces. The unpredictable nature of the dependences makes it difficult to replace the barriers by point-to-point synchronization at the granularity of bodies or cells, at least with the programming primitives we assume. The small number of barriers used in a time-step is independent of problem size or number of processors, depending only on the number of phases.

Synchronization is unnecessary within the force computation phase itself. While communication and sharing patterns in the application are irregular, they are phase structured. That is, although a process reads body and cell data from many other processes in the force calculation phase, the fields of a body structure that are written in this phase (the accelerations and velocities) are not the same as those that are read in it (the displacements and masses). The displacements are written only at the end of the update phase, and masses are not modified after initialization. However, in other phases, the program uses both mutual exclusion with locks and point-to-point event synchronization with flags in more interesting ways than Ocean. In the tree-building phase, a process that is ready to add a body to a cell must first obtain mutually exclusive access to the cell since other processes may want to read or modify the cell at the same time. This is implemented with a lock per cell. The phase that calculates cell centers of mass is essentially an upward pass through the tree from the leaves to the root, computing the moments of cells from those of their children. Point-to-point event synchronization is implemented using flags to ensure that a parent does not read the moment of its child until that child has itself been updated by all its children. This is an example of multiple-producer, single-consumer group synchronization. There is no synchronization within the update phase.

The work between synchronization points is large, particularly in the force computation and update phases, where it is

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

and $O(n/p)$, respectively. The need for locking cells in the tree-building and center-of-mass phases causes the work between synchronization points in those phases to be substantially smaller.

Mapping

The irregular nature makes this application more difficult to map perfectly for network locality in common networks such as meshes. The ORB partitioning scheme maps very naturally to a hypercube topology (discussed in Chapter 10) but not so well to a mesh or other less richly interconnected network. This property does not hold for costzones partitioning, which naturally maps to a one-dimensional array of processors but does not easily guarantee to keep communication local in most network topologies.

Summary

The Barnes-Hut application exhibits irregular, fine-grained, time-varying communication and data access patterns that are becoming increasingly prevalent even in scientific computing as we try to model more complex natural phenomena. Successful partitioning techniques for this application are not obvious by inspection of the code

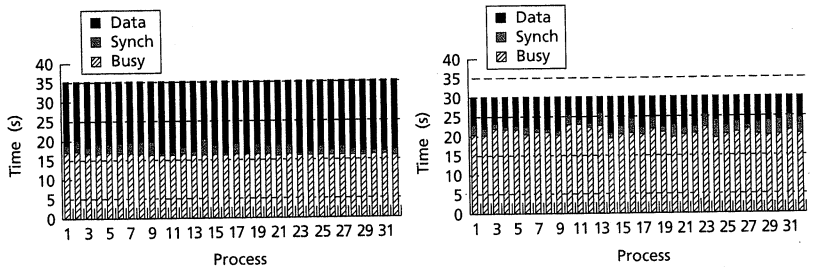


FIGURE 3.19 Execution time breakdown for Barnes-Hut with 512-K bodies on the Origin2000. The particular static assignment of bodies used is quite randomized, so given the large number of bodies relative to processors, the workload evens out due to the law of large numbers. The bigger problem with the static assignment is that because it is effectively randomized, the particles assigned to a processor are not close together in space so the communication-to-computation ratio is much larger than in the semistatic scheme. This is why data wait time is much smaller in the semistatic scheme. If we had assigned contiguous areas of space to processes statically, data wait time would be small, but load imbalance and hence synchronization wait time would be large. Even with the current static assignment, there is no guarantee that the assignment will remain load balanced as the galaxy evolves over time.

and require the use of insights from the application domain. These insights allow us to avoid using fully dynamic assignment methods, such as task queues and stealing.

Figure 3.19 shows the breakdown of execution time for this application on the 32-processor SGI Origin2000 machine. Load balance is quite good even with a static partitioning of the array of body pointers to processors precisely because there is little relationship between the locations of the bodies in the array and in physical space. However, the data access cost for a static partition is high due to a considerable amount of inherent and artifactual communication caused by the lack of contiguity in physical space. Semistatic costzone partitioning reduces this data access overhead substantially without compromising load balance.

3.5.3 Raytrace

Recall that in ray tracing rays are shot through the pixels in an image plane into a three-dimensional scene and the paths of the rays are traced as they bounce around to compute a color and opacity for the corresponding pixels. The algorithm uses a hierarchical representation of space called a Hierarchical Uniform Grid (HUG), which is similar in structure to the oct-tree used by the Barnes-Hut application. The root of the tree represents the entire space enclosing the scene, and each leaf holds a linked list of the object primitives that fall into that leaf (the maximum number of

primitives per leaf is defined by the user, as are some other aspects of the tree structure). The hierarchical grid or tree makes it efficient to skip empty regions of space when tracing a ray and quickly find the next interesting cell.

The Sequential Algorithm

For a given viewpoint, the sequential algorithm fires one ray into the scene through every pixel in the image plane. These initial rays are called primary rays. At the first object that a ray encounters (found by traversing the hierarchical uniform grid), it is first reflected toward every light source to determine whether it is in shadow from that light source. If it isn't, the contribution of the light source to its color and brightness is computed. The ray is also reflected from and refracted through the object as appropriate. Each reflection and refraction spawns a new ray, which undergoes the same procedure recursively for every object that it encounters. Thus, each primary ray generates a tree of rays. Rays are terminated when they leave the volume enclosing the scene or according to some user-defined criterion (such as the maximum number of levels allowed in a ray tree). Ray tracing, and computer graphics in general, affords several trade-offs between execution time and image quality, and many algorithmic optimizations have been developed to improve performance without significantly compromising image quality.

Decomposition and Assignment

There are two natural approaches to exploiting parallelism in ray tracing. One is to divide the space and, hence, the objects in the scene among processes and have a process compute the ray interactions that occur within its space. The unit of decomposition here is a subspace. When a ray leaves a process's subspace, it will be handled by the next process whose subspace it enters. This is called a *scene-oriented* approach. The alternative *ray-oriented* approach is to divide pixels in the image plane among processes. A process is responsible for the rays that are fired through its assigned pixels, and it follows a ray in its entire path through the scene, computing the interactions of the entire ray tree that the ray generates. The unit of decomposition here is a primary ray. The decomposition unit can be made finer by allowing different processes to process rays generated by the same primary ray (i.e., from the same ray tree) if necessary. The scene-oriented approach preserves more locality in the scene data since a process only touches the scene data in its subspace and the rays that enter that subspace. However, the ray-oriented approach is much easier to program—particularly starting from a sequential program that loops over rays—and to implement with low overhead in a shared address space since rays can be processed independently without synchronization and the scene data is read-only. It is also easily used in a message-passing model with explicit replication of nonlocal scene data. This program therefore uses a ray-oriented approach. The degree of concurrency for an n-by-n plane of pixels is $O(n^2)$ and is usually ample.

Unfortunately, a static block partitioning of the image plane would not be load balanced. Rays from different parts of the image plane might encounter very different numbers of reflections and hence very different amounts of work. The distribution of work is highly unpredictable, so we use a distributed task-queuing system (one queue per processor) with task stealing for load balancing.

To determine how to initially assign rays or pixels to processes, consider communication. Since the scene data is read-only, it causes no inherent communication. If we replicated the entire scene on every node, there would be no communication at all except due to task stealing. However, this approach does not allow us to render a scene larger than what fits in a single node's memory, so the data set size cannot scale with the number of processors used. Other than task stealing, communication is generated because only $1/p$ of the scene is allocated locally on a node while a process accesses the scene widely and unpredictably. To reduce this artifactual communication, we would like processes to reuse scene data as much as possible rather than to access the entire scene randomly. For this, we can exploit spatial coherence in ray tracing: because of the way light is reflected and refracted, rays that pass through adjacent pixels from the same viewpoint are likely to traverse similar parts of the scene and to be reflected in similar ways. This suggests that we should use domain decomposition on the image plane to initially assign pixels to task queues. Since the adjacency or spatial coherence of rays works in all directions in the image plane, a block-oriented domain decomposition works well. This also reduces the communication of image pixels themselves.

Given p processors, the image plane is partitioned into p rectangular blocks of size as close to equal as possible. Every image block or partition is further subdivided into fixed-size square image *tiles*, which are the units of task granularity and stealing (see Figure 3.20 for a four-process example). These tile tasks are initially inserted into the task queue of the processor to which that block is assigned. A processor ray traces the tiles in its block in scan-line order. When it has finished with its block, it steals tile tasks from other processors that are still busy. The choice of tile size is a compromise between preserving locality through spatial coherence and reducing the number of accesses to other processors' queues, both of which reduce communication, and keeping the task size small enough to ensure good load balance. We could also initially assign tiles to processes in an interleaved manner in both dimensions (called a *scatter decomposition*) to improve load balance in the initial assignment and reduce task stealing at some cost in spatial coherence.

Orchestration

Given the preceding decomposition and assignment, let us examine spatial locality, temporal locality, and synchronization.

Spatial Locality Most of the shared data accesses are to the scene data. However, because of changing viewpoints and the fact that rays bounce about unpredictably, it is impossible to divide the scene into parts that are each accessed only (or even dominantly) by a single process. In addition, the scene data structures are naturally small

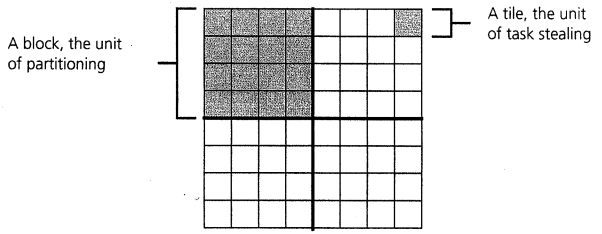


FIGURE 3.20 Image plane partitioning in Raytrace for four processors. Each tile contains several pixels. A contiguous block of tiles is assigned to every process. When a process has finished processing its assigned block, it steals available tiles from other processes.

and linked together with pointers, so it is very difficult to distribute them among memories at the granularity of pages. We therefore resort to using a round-robin layout of the pages that hold scene data to reduce hot spots and contention. Image data is small, and we try to allocate the few pages it falls on in different memories as with scene data. The block assignment described previously preserves spatial locality at cache block granularity in the image plane quite well, though it can lead to some loss of locality at tile boundaries, particularly with task stealing. A strip decomposition in rows of the image plane would be better from the viewpoint of spatial locality but would not exploit spatial coherence in the scene so well. As in Ocean, the best choices may be architecture dependent, and the assignment can be easily parameterized. Spatial locality on scene data is not very high and does not improve with larger scenes.

Temporal Locality Because of the read-only nature of the scene data, if there were unlimited capacity for replication, then only the first reference to nonlocally allocated data would cause communication. With finite replication capacity, on the other hand, data may be replaced and may have to be recommunicated. The domain decomposition and spatial coherence methods described earlier enhance temporal locality on scene data and reduce the sizes of the working sets. However, since the access patterns are so unpredictable due to the bouncing of rays, working sets are relatively large and ill defined. Note that most of the scene data accessed and hence the working sets are likely to be nonlocal. Nonetheless, this shared address space program does not replicate data in main memory: the working sets are not sharp, caches on machines are becoming larger, and replication in main memory has a cost, so it is unclear that the benefits outweigh the overhead.

Synchronization and Granularity Only a single barrier is used after an entire scene is rendered and before it is displayed. Locks are used to protect task queues and for some global variables that track statistics for the program. The work between synchronization points is the work associated with a tile of rays, which is usually quite large.

Mapping

Since Raytrace has very unpredictable access and communication patterns to its scene data, there is little scope for optimizing artifactual communication through mapping. The initial assignment partitions the image into a two-dimensional grid of blocks, making it natural to map to a two-dimensional mesh network, but the effect of mapping is not likely to be large.

Summary

This application tends to have large working sets and relatively poor spatial locality but a low communication-to-computation ratio as long as there is ample scene replication capacity. Figure 3.21 shows the breakdown of execution time on the Origin2000 machine for a standard data set consisting of a number of balls arranged in a bunch, illustrating the importance of task stealing in reducing load imbalance and hence wait time at barrier synchronization. The extra communication and synchronization incurred as a result of task stealing is well worthwhile.

3.5.4 Data Mining

A key difference in the data mining application from the previous ones is that the data being accessed and manipulated typically resides on disk rather than in memory. It is very important to reduce the number of disk accesses since their cost is very high and to reduce the contention for a disk controller by different processors. The techniques for reducing disk access cost are essentially the same as those for reducing communication and memory access cost.

Recall the basic insight used in association mining: if an itemset of size k is large (i.e., it occurs in more than a threshold fraction of the transactions), then all subsets of that itemset must also be large. For illustration, consider a database consisting of five items—A, B, C, D, and E—of which one or more may be present in a particular transaction. The items within a transaction are lexicographically sorted. Consider L_2, the list of large itemsets of size two. This list might be {AB, AC, AD, BC, BD, CD, DE}. The itemsets within L_2 are also lexicographically sorted. Given this L_2, the list of itemsets that are candidates for membership in L_3 are obtained by performing a *join* operation on the itemsets in L_2—that is, taking pairs of itemsets in L_2 that share a common first item (say, AB and AC) and combining them into a lexicographically sorted itemset of size three (here ABC). The resulting candidate list C_3 in this case is {ABC, ABD, ACD, BCD}. Of these itemsets in C_3, some may actually occur with enough frequency to be placed in L_3, and so on. In general, the join operation to obtain C_k from L_{k-1} finds pairs of itemsets in L_{k-1} whose first $k-2$ items are the same and combines them to create a new itemset for C_k. Itemsets of size $k-1$ that have common $(k-2)$-sized prefixes are said to form an equivalence class (e.g., {AB, AC, AD}, {BC, BD}, {CD}, and {DE} in this example for $k = 3$). Only itemsets in the same $k-2$ equivalence class need to be considered together to form C_k from L_{k-1}, which greatly reduces the number of pairwise itemset comparisons we need to do to determine C_k.

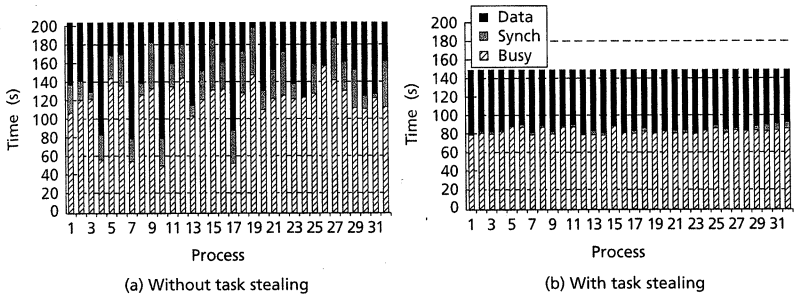


FIGURE 3.21 Execution time breakdown for Raytrace with the balls data set on the Origin2000. Task stealing is clearly very important for balancing the workload (and hence reducing synchronization wait time at barriers) in this highly unpredictable application.

The Sequential Algorithm

A simple sequential method for association mining is to first traverse the data set and record the frequencies of all itemsets of size one, thus determining L_1. From L_1, we can construct the candidate list C_2 and then traverse the data set again to find which entries of C_2 occur frequently enough to be placed in L_2. From L_2, we can construct C_3 and then traverse the data set to determine L_3, and so on until we have found L_k. Although this method is simple, it requires reading all transactions in the database from disk k times, which is expensive.

An alternative sequential algorithm seeks to reduce the amount of work done to compute candidate lists C_k from lists of large itemsets L_{k-1} and especially to reduce the number of times data must be read from disk to determine the frequencies of the itemsets in C_k. We have seen that equivalence classes can be used to achieve the first goal. In fact, they can be used to construct a method that achieves both goals together. The idea is to transform the way in which data is stored in the database. Instead of storing transactions in the form {T_x, A, B, D, . . .}—where T_x is the transaction identifier and A, B, D are items in the transaction—we can keep in the database records of the form {IS_x, T_1, T_2, T_3, . . .}, where IS_x is an itemset and T_1, T_2, and so on are transactions that contain that itemset. That is, a database record is maintained per itemset rather than per transaction. If the large itemsets of size $k-1$ (i.e., the elements of L_{k-1}) that are in the same $k-2$ equivalence class are identified, then

computing the candidate list C_k requires only examining all pairs of these itemsets. Since each itemset has its list of transactions attached, the size of each resulting itemset in C_k can be computed at the same time as constructing the C_k itemset itself from a pair of L_{k-1} itemsets, by simply computing the intersection of the transactions in that pair's lists.

EXAMPLE 3.4 Suppose {AB, 1, 3, 5, 8, 9} and {AC, 2, 3, 4, 8, 10} are large itemsets of size two in the same one-equivalence class (they each start with A). How will the data be accessed in disk and memory?

Answer The list of transactions that contain itemset ABC is {3, 8}, so the occurrence count of itemset ABC is two. This means that once the database is transposed and the one-equivalence classes identified, the rest of the computation for a single one-equivalence class can be done to completion (i.e., all large itemsets of size *k* found) before considering any data from other one-equivalence classes. If a one-equivalence class fits in main memory, then after the transposition of the database a given data item needs to be read from disk only once, greatly reducing the number of expensive I/O accesses. A form of blocking for temporal locality has been achieved. ■

Decomposition and Assignment

The two sequential methods also differ in their parallelization, with the latter method having advantages in this respect as well. To parallelize the first method, we could first divide the database among processors. At each step, a processor traverses only its local portion of the database to determine partial occurrence counts for the candidate itemsets, incurring no communication or nonlocal disk accesses in this phase. The partial counts are then merged into global counts to determine which of the candidates are large. Thus, in parallel this method requires not only multiple passes over the database but also interprocessor communication and synchronization at the end of every pass.

In the second method, the equivalence classes that helped the sequential method reduce disk accesses are very useful for parallelization as well. Since the computation on each one-equivalence class is independent of the computation on any other, we can simply divide the one-equivalence classes among processes that can thereafter proceed independently for the rest of the program without communication or synchronization. The itemset lists (in the transformed format) corresponding to an equivalence class can be stored on the local disk of the process to which the equivalence class is assigned, so no need for remote disk access remains after this point. As in the sequential algorithm, each process can complete the work on one of its assigned equivalence classes before proceeding to the next one, so each itemset record from the local disk should also be read only once as part of its equivalence class.

The challenge is ensuring a load-balanced assignment of equivalence classes to processes. A simple metric for load balance is to assign equivalence classes based on

the number of initial entries in them. However, as the computation unfolds to compute itemsets of size *k*, the amount of work is determined more closely by the number of large itemsets that are generated at each step. Heuristic measures that estimate this or some other more appropriate work metric can be used as well. Otherwise, we may have to resort to dynamic tasking and task stealing, which can compromise much of the simplicity of this method (namely, that once processes are assigned their initial equivalence classes, they no longer have to communicate, synchronize, or perform remote disk access).

The first step in this approach, of course, is to compute the one-equivalence classes and the large itemsets of size two in them as a starting point for the parallel assignment. To compute these itemsets, we are better off using the original transaction-oriented form of the database rather than the transformed version, so we do not transform the database yet (see Exercise 3.18). Every process sweeps over the transactions in its local portion of the database and, for each pair of items in a transaction, increments a local counter for that item pair (the local counts can be maintained as a two-dimensional upper-triangular matrix, with the indices being items). The local counts are then merged, involving interprocess communication, and the large itemsets of size two are determined from the resulting global counts. These itemsets are then partitioned into one-equivalence classes, which are assigned to processes as described earlier.

The next step is to transform the database from the original $\{T_x, A, B, D, \ldots\}$ organization by transaction to the $\{IS_x, T_1, T_2, T_3, \ldots\}$ organization by itemset, where the IS_x are initially the size two itemsets. This can be done in two steps—a local step and a communication step. In the local step, a process constructs the partial transaction lists for large itemsets of size two from its local portion of the database. In the communication step, a process (at least conceptually) "sends" the lists for those size two itemsets whose one-equivalence classes are not assigned to it to the process to which they are assigned and "receives" from other processes the lists for the equivalence classes that are assigned to it. The incoming partial lists are merged into the local lists, preserving a lexicographically sorted order, after which the process holds the transformed database for its assigned equivalence classes. It can now compute the itemsets of size *k* step by step for each of its equivalence classes, without any communication, synchronization, or remote disk access (if there is no task stealing). At the end of the calculation, the results for the large itemsets of size *k* are available from the different processes. The communication step of the transformation phase is usually the most expensive step in the algorithm and is quite like transposing a matrix, except that the sizes of the communications among different pairs of processes are different.

Orchestration

Given this decomposition and assignment, let us examine spatial locality, temporal locality, and synchronization.

Spatial Locality The organization of the computation and the lexicographic sorting of the itemsets and transactions causes most of the traversals through the data to be simple front-to-back sweeps that exhibit very good predictability and spatial locality. This is particularly important in reading from disk, since it is important to amortize the high start-up costs of a disk read over a large amount of useful data.

Temporal Locality As discussed earlier, proceeding over one equivalence class at a time is much like blocking, although how successful it is depends on whether the data for that equivalence class fits in main memory. As the computation for an equivalence class proceeds, the number of large itemsets becomes smaller, so reuse in main memory is more likely to be exploited. Note that here it is more likely that we are exploiting temporal locality in main memory rather than in the cache, although the techniques and goals are similar at any level of the extended memory hierarchy.

Synchronization The major forms of synchronization are the reductions of partial occurrence counts into global counts in the first step of the algorithm (computing the large size two itemsets) and a barrier after this to begin the transformation phase. The reduction is required only for itemsets of size two since thereafter every process continues independently to compute the large itemsets of size k in its assigned equivalence classes. Further synchronization may be needed if dynamic task management is used for load balancing.

Mapping

The communication to transform the database is all-to-all: a process may "send" different itemsets of size two and their partial transaction lists to all other processes and may "receive" or read such lists from them all. It is difficult to map all-to-all communication in a contention-free manner to network topologies (like meshes or rings) that are not very richly interconnected. Endpoint contention is reduced by communication scheduling techniques such as having each processor i at step j exchange data with processor i `xor` j so that no processor or node is overloaded.

Summary

Data mining differs from the other application case studies since disk access is a major bottleneck, and parallelization techniques aim primarily to minimize its cost. The technique we have examined treats the disk as simply another, explicitly managed level of the extended memory hierarchy. Load balance is an outstanding question that can compromise some of the local properties of the parallel program.

3.6 IMPLICATIONS FOR PROGRAMMING MODELS

We have seen throughout this and the previous chapter that while the decomposition and assignment of a parallel program are often (but not always) independent of the programming model, the orchestration step is highly dependent on it. In Chapter 1, we learned about the fundamental design issues that apply to any layer of the communication architecture, including the programming model. We learned that the two major programming models—a shared address space and explicit message passing between private address spaces—are fundamentally distinguished by functional differences, such as naming, replication, and synchronization. While either programming model can be implemented on any communication abstraction and hardware, the positions taken on these functional issues at a given layer affect (and are influenced by) performance characteristics, such as overhead, latency, and bandwidth. At that stage, we only dealt with those issues in the abstract and could not appreciate the interactions with applications and the implications regarding which programming models are preferable under what circumstances. Now that we have an in-depth understanding of several interesting parallel applications and understand the performance issues in orchestration, we can compare the programming models in light of application and performance characteristics.

We will use the application case studies to illustrate the issues, assuming a generic multiprocessor architecture with physically distributed memory. For a shared address space, we assume that read (loads) and write (stores) to shared data are the only communication mechanisms exported to the user, and we call this a read-write shared address space. Of course, in practice nothing stops a system from providing support for explicit messages as well as these primitives in a shared address space model, but we ignore this possibility for now. The shared address space model can be supported in a wide variety of ways at the communication abstraction and hardware/software interface (recall the discussion of naming models at the end of Chapter 1) with different granularities and different efficiencies for supporting communication, replication, and coherence. These affect the success of the programming model and will be discussed in detail in Chapters 8 and 9. Here we focus on the most common case in which a cache-coherent shared address space is supported efficiently at fine granularity—for example, with direct hardware support for a shared physical address space as well as for communication, replication, and coherence at the fixed granularity of cache blocks. However, contrast this common case with a hardware-supported shared address space without coherent replication, as provided by the BBN Butterfly and CRAY T3D and T3E machines. For the message-passing programming model, we will assume that it too is supported efficiently by the communication abstraction and the hardware/software interface.

As application programmers, we view the programming model as our window to the communication architecture. Differences between programming models and how they are implemented have implications for ease of programming, for the structuring of communication, for performance, and for scalability. In addition to functional aspects (like naming, replication, and synchronization), there are organizational aspects (like the granularity at which communication is performed) and performance aspects (like the endpoint overhead of a communication operation) that differ across programming models and affect programming for performance. Other performance aspects (such as latency and available bandwidth) depend